CSCE 750: Analysis of Algorithms
Course Notes

Stephen A. Fenner

Lecture 1
Introduction to Algorithms Analysis

I'm assuming you've all had CSCE 350 or the equivalent. I'll assume some basic things from there, and sometimes quickly review the more important/subtle points.

We'll start with Appendix A in CLRS — Summations. Why? They are essential tools in analyzing the complexity of algorithms.

A First Example

Consider the following two C code fragments:

```
/* Fragment 1 */
sum = 0;
for (i=1; i<n; i*=2)
    for (j=0; j<i; j++)
        sum++;
```

and

```
/* Fragment 2 */
sum = 0;
for (i=1; i<n; i*=2)
    for (j=i; j<n; j++)
        sum++;
```

Note the subtle difference. Is there a difference in running time (order-of-magnitude as a function of n)?

Yes there is. Sample 1 runs in time $\Theta(n)$ and Sample 2 runs in time $\Theta(n \log n)$, so Sample 2 runs *significantly* longer.

[Recall:

- $f = O(g)$ means $f(n)$ is at most a constant times $g(n)$ for all n large enough.
- $f = \Omega(g)$ means $f(n)$ is at least a (positive) constant times $g(n)$ for all n large enough. (Equivalently, $g = O(f)$.)

- $f = \Theta(g)$ means both $f = O(g)$ and $f = \Omega(g)$. "$f = \Theta(g)$" is an equivalence relation between f and g.

Also, $\log n$ means $\log_2 n$.]

Here's the intuition: in both fragments, the variable i does not run from 1 to n at an even pace. Since it doubles each time, it spends most of its time being very small compared to n, which makes the first j-loop run faster and the second j-loop run slower.

Let's analyze the running times more rigorously. We generally don't care about constant factors, so it is enough to find, for each fragment, an upper bound and a lower bound that are within a constant factor of each other. This looseness usually makes life a lot easier for us, since we don't have to be exact.

Claim 1 *The running time for Fragment 2 is $O(n \log n)$.*

Proof The body of the inner loop (j-loop) takes $O(1)$ time. Each time it runs, the j-loop iterates $n - i \leq n$ times, for a time of $O(n)$ per execution. The outer i-loop runs about $\log n$ many times (actually, exactly $\lceil \log n \rceil$ many times). So the total time for the fragment (including initialization, loop testing, and increment) is $O(n \log n)$. □

Claim 2 *The running time for Fragment 2 is $\Omega(n \log n)$.*

Proof Note that for all iterations of the i-loop except the last one, the j-loop iterates at least $n/2$ times (because $i < n/2$ and thus $n - i > n - n/2 = n/2$). Thus the `sum` variable is incremented at least $\frac{n}{2} \times (\log n - 1)$ times total, which is clearly $\Omega(n \log n)$. □

Claim 3 *The running time for Fragment 1 is $\Omega(n)$.*

Proof To get a lower bound, we only need to look at the last iteration of the i-loop! (Digression: the late Paul Erdős, arguably the greatest mathematician of the 20th century, once described the art of mathematical analysis as "knowing what information you can throw away.") The value of i in the last i-loop iteration must be at least $n/2$. In this iteration (as in all iterations), j runs from 0 through $i - 1$, so the `sum` variable is incremented $i \geq n/2$ times. □

But maybe we threw too much away in ignoring the previous i-loop iterations, so that our lower bound of $\Omega(n)$ is not tight enough. Actually, it is tight.

Claim 4 *The running time for Fragment 1 is $O(n)$.*

Proof Now we must be more careful; the simple analysis that worked for Fragment 2 is not good enough here. We must bite the bullet and take a sum of the running times of the j-loop as i increases. For all $0 \leq k < \lceil \log n \rceil$, during the $(k+1)$st iteration of the i-loop we clearly have $i = 2^k$. For this $(k+1)$st iteration of the i-loop, the j-loop iterates $i = 2^k$ times, so it takes time at most $C \cdot 2^k$ for some constant $C > 0$. So the total time for the fragment is

$$T \leq \sum_{k=0}^{\lceil \log n \rceil - 1} C \cdot 2^k.$$

This is a finite geometric series (see below) whose exact value is

$$C \cdot \frac{2^{\lceil \log n \rceil} - 1}{2 - 1} = C \cdot (2^{\lceil \log n \rceil} - 1).$$

But $\lceil \log n \rceil < \log n + 1$, so

$$T \leq C \cdot (2^{\log n + 1} - 1) < C \cdot 2^{\log n + 1} = 2Cn,$$

which is $O(n)$ as claimed. Even if we add the extra bits (initialization, loop testing, incrementing), these can be "absorbed" into the $O(n)$ bound. □

*head*Series

Geometric Series

Geometric series come in two flavors: finite and infinite. A finite geometric series is of the form

$$\sum_{i=0}^{n-1} r^i,$$

where r is a constant (real, but could also be complex), and n is a nonnegative integer. If $r = 1$, then this sum is clearly equal to n. To find a closed form when $r \neq 1$, let S be the value of the sum above. Then

$$\begin{aligned} S - rS &= \sum_{i=0}^{n-1} r^i - r \sum_{i=0}^{n-1} r^i \\ &= \sum_{i=0}^{n-1} r^i - \sum_{i=0}^{n-1} r^{i+1} \\ &= \sum_{i=0}^{n-1} r^i - \sum_{i=1}^{n} r^i \\ &= 1 - r^n. \end{aligned}$$

The last equation holds because most of the terms in the two sums cancel. This is called telescoping. Dividing by $1 - r$ (which is not zero!), we get

$$S = \frac{1 - r^n}{1 - r}.$$

An infinite geometric series has the form

$$\sum_{i=0}^{\infty} r^i,$$

which is the limit of the finite geometric series as $n \to \infty$. We have

$$\begin{aligned} \sum_{i=0}^{\infty} r^i &= \lim_{n \to \infty} \sum_{i=0}^{n-1} r^i \\ &= \lim_{n \to \infty} \frac{1 - r^n}{1 - r} \\ &= \frac{1}{1 - r} - \frac{1}{1 - r} \lim_{n \to \infty} r^n. \end{aligned}$$

This last limit exists if and only if $|r| < 1$, in which case it is zero. So for $|r| < 1$, the infinite geometric series above has value $1/(1-r)$.

Lecture 2
More About Sums

Arithmetic Series

How to compute a closed form for $\sum_{i=1}^{n} i$? This method is attributed to Gauss as a child: Let $S = \sum_{i=1}^{n} i$. Then reversing the terms, we see that $S = \sum_{i=1}^{n}(n+1-i)$. Then adding these two sums term by term, we get

$$2S = \sum_{i=1}^{n}(i + (n+1-i)) = \sum_{i=1}^{n}(n+1) = n(n+1),$$

so $S = n(n+1)/2$. Notice that this means that $\sum_{i=1}^{n} i = \Theta(n^2)$.

Higher Powers

Closed forms can be computed for $\sum_{i=1}^{n} i^k$ for any integer $k \geq 0$. We have $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$ and $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$. More generally, we know that

$$\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$$

for all integers $k \geq 0$.

Supplement: Finding Closed Forms for $\sum_{i=0}^{n-1} i^k$

The sum $\sum_{i=0}^{n-1} i^k$ has a closed form for every integer $k \geq 0$. The closed form is always a polynomial in n of degree $k+1$. Here's how to find it using induction on k. For each i and $k \geq 0$ define $i^{\underline{k}} := i(i-1)(i-2)\cdots(i-k+1)$ (k many factors). First, we'll find a closed form for the sum $\sum_{i=0}^{n-1} i^{\underline{k}}$, and from this we can easily get a closed form for $\sum_{i=0}^{n-1} i^k$. Consider the sum

$$S := \sum_{i=0}^{n-1} \left[(i+1)^{\underline{k+1}} - i^{\underline{k+1}}\right].$$

This sum telescopes in two different ways. First of all, we clearly have that $S = n^{\underline{k+1}} - 0^{\underline{k+1}} = n^{\underline{k+1}}$. Second, each term of the sum can be factored:

$$\begin{aligned}
\sum_{i=0}^{n-1} \left[(i+1)^{\underline{k+1}} - i^{\underline{k+1}}\right] &= \sum_{i=0}^{n-1} \left[((i+1)i(i-1)\cdots(i-k+1)) - (i(i-1)\cdots(i-k+1)(i-k))\right] \\
&= \sum_{i} \left[(i+1) - (i-k)\right] i(i-1)\cdots(i-k+1) \\
&= (k+1)\sum_{i} i^{\underline{k}}.
\end{aligned}$$

Equating these two expressions for S and dividing by $k+1$, we get

$$\sum_{i=0}^{n-1} i^{\underline{k}} = \frac{n^{\underline{k+1}}}{k+1} = \frac{1}{k+1} n(n-1)(n-2)\cdots(n-k),$$

which is a closed form for $\sum_{i=0}^{n-1} i^{\underline{k}}$. Now to get a closed form for $\sum_{i=0}^{n-1} i^k$, we first expand the product $i^{\underline{k}}$ completely. For example, for $k=2$ we would get $i^{\underline{2}} = i(i-1) = i^2 - i$. We then sum this for i going from 0 to $n-1$ and use the closed form we got above:

$$\sum_{i=0}^{n-1} (i^2 - i) = \sum_i i^{\underline{2}} = \frac{1}{2+1} n(n-1)(n-2) = \frac{n(n-1)(n-2)}{3}.$$

The left-hand side is equal to $\sum_{i=0}^{n-1} i^2 - \sum_{i=0}^{n-1} i$. We already know the second term—it is $n(n-1)/2$, and so we move it to the other side:

$$\sum_{i=0}^{n-1} i^2 = \frac{n(n-1)(n-2)}{3} + \frac{n(n-1)}{2} = \frac{n(n-1)(2n-1)}{6}.$$

So we have a closed form for the sum of squares. For more general k, when we expand $i^{\underline{k}}$ and sum, we'll get some combination $\sum_i i^k + \cdots$, where the other terms all involve sums of smaller powers of i. Assuming by induction that we have closed forms for all the smaller power sums, we then get a closed form for $\sum_i i^k$.

As an exercise, apply the ideas above to find a closed form for $\sum_{i=0}^{n-1} i^3$.

Harmonic Series

What about $k=-1$? We show that $\sum_{i=1}^{n} 1/i = \Theta(\log n)$ (more accurately, $\sum_{i=1}^{n} 1/i = \ln n + O(1)$). We use the technique of nested sums. Assume $n \geq 1$ and let $m = \lfloor \log(n+1) \rfloor$, that is, m is the largest integer such that $2^m \leq n+1$. Then,

$$\sum_{i=i}^{n} \frac{1}{i} = \sum_{b=0}^{m-1} \sum_{i=0}^{2^b-1} \frac{1}{2^b+i} + \sum_{i=2^m}^{n} \frac{1}{i}. \tag{1}$$

The inner sum on the right-hand side gives a block of contiguous terms in the sum on the left-hand side. The last sum on the right gives the terms left over after all the complete blocks are counted.

We use (1) to get both lower and upper bounds on the harmonic series. For the upper bound, we see that

$$\sum_{b=0}^{m-1} \sum_{i=0}^{2^b-1} \frac{1}{2^b+i} \leq \sum_{b=0}^{m-1} \sum_{i=0}^{2^b-1} \frac{1}{2^b} = \sum_{b=0}^{m-1} 1 = m,$$

and

$$\sum_{i=2^m}^{n} \frac{1}{i} \leq \sum_{i=2^m}^{n} \frac{1}{2^m} \leq 1,$$

since there are at most 2^m terms in this sum. Thus $\sum_{i=1}^{n} 1/i \leq m+1 = O(\log n)$.

We do something similar for the lower bound.

$$\sum_{b=0}^{m-1}\sum_{i=0}^{2^b-1}\frac{1}{2^b+i} \geq \sum_{b=0}^{m-1}\sum_{i=0}^{2^b-1}\frac{1}{2^{b+1}} = \sum_{b=0}^{m-1}\frac{1}{2} = \frac{m}{2},$$

so $\sum_{i=1}^{n} 1/i \geq m/2 = \Omega(\log n)$.

Differentiating and Integrating Series

Closed forms for some series can be obtained by differentiating or integrating term-by-term another series with a known closed form. For example, consider the series

$$\sum_{i=1}^{n} ir^i, \tag{2}$$

for some fixed $r \neq 1$. There are at least two ways for finding a closed form for (2).

Direct Method: Set $S = \sum_{i=1}^{n} ir^i$. Then, as in the case of the geometric series, we have

$$\begin{aligned}
S - rS &= \sum_{i=1}^{n} ir^i - \sum_{i=1}^{n} ir^{i+1} \\
&= \sum_{i=1}^{n} ir^i - \sum_{i=0}^{n} ir^{i+1} \\
&= \sum_{i=1}^{n} ir^i - \sum_{i=1}^{n+1}(i-1)r^i \\
&= \sum_{i=1}^{n} ir^i - \sum_{i=1}^{n+1} ir^i + \sum_{i=1}^{n+1} r^i \\
&= -(n+1)r^{n+1} + r\sum_{i=0}^{n} r^i \\
&= -(n+1)r^{n+1} + \frac{r(1-r^{n+1})}{1-r},
\end{aligned}$$

so

$$\sum_{i=1}^{n} ir^i = \frac{r(1-r^{n+1})}{(1-r)^2} - \frac{(n+1)r^{n+1}}{1-r}. \tag{3}$$

Differentiation Method: Differentiate both sides of the equation

$$\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$$

with respect to r. By linearity of d/dr, we get

$$\sum_{i=0}^{n} ir^{i-1} = \frac{-(n+1)r^n(1-r) + (1-r^{n+1})}{(1-r)^2}.$$

Multiplying both sides by r, ignoring the $i = 0$ term on the left, and rearranging the right gives (3).

Taking the limit as $n \to \infty$ in (3), we get, for all r such that $|r| < 1$,

$$\sum_{i=1}^{\infty} ir^i = \frac{r}{(1-r)^2}.$$

Note that $(n+1)r^{n+1}$ tends to zero, because $n+1$ increases only linearly in n, but r^{n+1} decreases exponentially in n.

Integrating a series term by term can also be useful.

Exercise Find a closed form for the series $\sum_{i=1}^{\infty} \frac{r^i}{i}$, where $|r| < 1$. [Hint: integrate the infinite geometric series term by term. Doing this is legal because the series $\sum_{i=1}^{\infty} \left| \frac{x^i}{i} \right|$ converges uniformly for x in some neighborhood of r.]

Lecture 3
More Math

Products

A product is like a sum, but the terms are multiplied instead of added. For example,

$$n! = \prod_{i=1}^{n} i.$$

By convention, an empty sum has value 0, and an empty product has value 1. So in particular,

$$0! = \prod_{i=1}^{0} i = 1.$$

There are tricks for getting a closed form for a product, but a good fall-back method is to just take the logarithm (to any convenient base b). This turns the product into a sum. Simplify the sum, then take b to the power of the result, then simplify further if possible. For example, suppose we want to evaluate

$$S = \prod_{i=0}^{n} 2 \cdot 3^i.$$

Taking the log (to base two) of both sides we have

$$\begin{aligned} \log S &= \sum_{i=0}^{n} \log(2 \cdot 3^i) \\ &= \sum_{i=0}^{n} (\log 2 + i \log 3) \\ &= n + 1 + \log 3 \sum_{i=0}^{n} i \\ &= n + 1 + \log 3 \frac{n(n+1)}{2}. \end{aligned}$$

Taking 2 to the power of each side, we get

$$\begin{aligned} S &= 2^{n+1} \cdot 2^{(\log 3)n(n+1)/2} \\ &= 2^{n+1} \cdot (2^{\log 3})^{n(n+1)/2} \\ &= 2^{n+1} \cdot 3^{n(n+1)/2}. \end{aligned}$$

A quicker (but essentially equivalent) way to get the same closed form is to split the product, then use the fact that the product of b to the power of some term is equal to b to the power of the sum of the terms. So we have

$$\begin{aligned} \prod_{i=0}^{n} 2 \cdot 3^i &= \prod_{i=0}^{n} 2 \cdot \prod_{i=0}^{n} 3^i \\ &= 2^{n+1} \cdot 3^{\sum_{i=0}^{n} i} \\ &= 2^{n+1} \cdot 3^{n(n+1)/2}, \end{aligned}$$

as we had before.

Bounding Sums

We've already done a few of these. Here's one more: we show that $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$ for all fixed $k \geq 0$.

For the upper bound, we have

$$\sum_{i=1}^{n} i^k \leq \sum_{i=1}^{n} n^k = n^{k+1}.$$

For the lower bound, we have

$$\begin{aligned} \sum_{i=1}^{n} i^k &\geq \sum_{i=\lceil n/2 \rceil}^{n} i^k \\ &\geq \sum_{i=\lceil n/2 \rceil}^{n} \left(\frac{n}{2}\right)^k \\ &\geq \frac{n}{2} \left(\frac{n}{2}\right)^k \\ &= \frac{n^{k+1}}{2^{k+1}}. \end{aligned}$$

Integral Approximation

Sums can be approximated closely by integrals in many cases.

Definition 5 Let $I \subseteq \mathbb{R}$ be an interval with endpoints $a < b$ (a could be $-\infty$, and b could be ∞). A function $f : I \to \mathbb{R}$ is *monotone increasing on I* if, for all $x, y \in I$, if $x \leq y$ then $f(x) \leq f(y)$. The function f is *monotone decreasing on I* if we have $x \leq y$ implies $f(x) \geq f(y)$.

Theorem 6 *Let I, a, and b be in Definition 5, and let $f : I \to \mathbb{R}$ be some function integrable on I. For any integers c and d such that $a \leq c - 1 < d + 1 \leq b$, we have*

$$\int_{c-1}^{d} f(x)dx \leq \sum_{i=c}^{d} f(i) \leq \int_{c}^{d+1} f(x)dx$$

if f is monotone increasing on I, and

$$\int_c^{d+1} f(x)dx \leq \sum_{i=c}^{d} f(i) \leq \int_{c-1}^{d} f(x)dx$$

if f is monotone decreasing on I.

Proof Define $g(x) = f(\lfloor x \rfloor)$ and $h(x) = f(\lceil x \rceil)$. Assume first that f is monotone increasing. Then $g(x) \leq f(x) \leq h(x)$ for all $c-1 \leq x \leq d+1$. Thus we have

$$\begin{aligned}
\int_{c-1}^{d} f(x)dx &\leq \int_{c-1}^{d} h(x)dx \\
&= \sum_{i=c}^{d} f(i) \\
&= \int_{c}^{d+1} g(x)dx \\
&\leq \int_{c}^{d+1} f(x)dx,
\end{aligned}$$

which proves the first two inequalities in the theorem. Now if f is monotone decreasing, then $g(x) \geq f(x) \geq h(x)$, and so we get the same chain of inequalities as above but with the sense of the inequalities reversed. This proves the second two inequalities in the theorem. □

It is sometimes useful to know how tight the integral approximation is. This is done by computing the difference between the upper- and lower-bounding integrals. For the monotone increasing case (the decreasing case is similar), we have

$$\begin{aligned}
&\int_c^{d+1} f(x)dx - \int_{c-1}^{d} f(x)dx \\
&= \int_d^{d+1} f(x)dx - \int_{c-1}^{c} f(x)dx \\
&\leq f(d+1) - f(c-1).
\end{aligned}$$

This is usually tight enough for our purposes.

Let's redo our estimation of $\sum_{i=1}^{n} i^k$ using the integral approximation method. We can get our two constants much closer together in this case. Since $f(x) = x^k$ is monotone increasing on $[0, \infty)$ for $k \geq 0$, we use the first inequalities of Theorem 6 to get

$$\frac{n^{k+1}}{k+1} \leq \sum_{i=1}^{n} i^k \leq \frac{(n+1)^{k+1} - 1}{k+1}.$$

So we get not only that $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$ but also that we can choose our constants $c_1 = 1/(k+1)$ and c_2 to be anything greater than $1/(k+1)$, provided we then choose our threshold n_0 large enough.

Math Odds and Ends:

More Asymptotic Notation

We've seen O-, Ω-, and Θ-notation. There are two more o- (little "oh") and ω- (little omega) notation.

$f(n) = o(g(n))$ means that $f(n)$ grows strictly more slowly than *any* positive constant times $g(n)$. That is, for any $c > 0$ there is an n_0 such that

$$f(n) < c \cdot g(n)$$

for all $n \geq n_0$. Equivalently,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

$f(n) = o(g(n))$ implies $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$, but is strictly stronger than both combined.

$f(n) = \omega(g(n))$ means that $f(n)$ grows strictly faster than *any* positive constant times $g(n)$. That is, for any $c > 0$ there is an n_0 such that

$$f(n) > c \cdot g(n)$$

for all $n \geq n_0$. Equivalently,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty,$$

or equivalently, $g(n) = o(f(n))$. $f(n) = \omega(g(n))$ implies $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$, but is strictly stronger than both combined.

Modular Arithmetic

For integers a and n with $n > 0$, we define $a \bmod n$ to be the remainder after dividing a by n. That is,

$$a \bmod n = a - \lfloor a/n \rfloor n.$$

If $a \bmod n = b \bmod n$, then we may write

$$a \equiv b \pmod{n}.$$

This is true if and only if n is a divisor of $a - b$.

Exponentials

$$\lim_{n\to\infty} \frac{n^b}{a^n} = 0$$

for any real constants a and b with $a > 1$. That is, $n^b = o(a^n)$.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$

In particular, $e^x \geq 1 + x$ for any real x. We also have

$$e^x = \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n.$$

Two useful identities are

- $1 + x \leq e^x$ for all $x \in \mathbb{R}$.

- $e^x \leq 1 + x + x^2$ for all $x \in \mathbb{R}$ such that $|x| \leq 1$.

Logarithms
For all real $a > 0$, $b > 0$, $c > 0$ and n,

$$\begin{aligned} a &= b^{\log_b a}, \\ \log_c(ab) &= \log_c a + \log_c b, \\ \log_b a^n &= n \log_b a, \\ \log_b a &= \frac{\log_c a}{\log_c b}, \\ \log_b(1/a) &= -\log_b a, \\ \log_b a &= \frac{1}{\log_a b}, \\ a^{\log_b c} &= c^{\log_b a}, \end{aligned}$$

where we assume none of the logarithm bases are 1.

The natural logarithm $\ln x = \log_e x = \int_1^x dr/r$ satisfies

- $\ln(1 + x) = x - x^2/2 + x^3/3 - x^4/4 + \cdots = -\sum_{i=1}^{\infty} (-1)^i x^i / i$ (Taylor series),
- $\ln x \leq x - 1$ for all $x > 0$.

In mathematics, $\ln x$ is often denoted as $\log x$. We will use $\lg x$ to denote $\log_2 x$.

Lecture 4
Starting Algorithms

Problems, Algorithms, and the RAM Model

Types of problems: decision, search, arrangement, optimization. We have examples for each, and there can be overlap.

An algorithm is a careful, step-by-step procedure for performing some task. It should allow no room for judgment or intuition.

To analyze algorithms rigorously, we must express them concretely as programs on a completely specified mathematical model of computation. One such popular model (which is essentially equivalent to several others) is the RAM (Random Access Machine) model. Defining this model rigorously would be too tedious, so we'll only describe it. A RAM algorithm is a sequence of RAM instructions that act on (read and write) individual registers in a one-way infinite array (indexed $0, 1, 2, \ldots$). Each register is a finite sequence of bits. Typical allowed instructions include arithmetic (addition, substraction, multiplication, and division) on registers (both integer and floating point), loading and storing values from/in registers, indirect addressing, testing (equality, comparison) of registers (both integer and floating point), and branching. Thus the RAM model looks like some generic assembly language.

We count each instruction execution as costing one unit of time. This is only realistic, however, if we limit the range of possible values that can be stored in a register. A restriction that is commonly agreed upon is that for problems of size n, each register is allowed to be only $c \log n$ bits, where $c \geq 1$ is some constant that we can choose as big as we need for the problem at hand, so

long as it is independent of n. If we didn't restrict the size of registers, then we could manipulate huge amounts of data in a single register in a single time step (say, by repeated squaring). This is clearly unrealistic.

Sometimes it is unclear whether or not we should allow a particular operation as a (primitive) instruction in the RAM model. For example, exponentiation. Generally, exponentiation is not considered a primitive operation, however, most architectures allow arithmetic shifts by k bits, which is essentially multiplication by 2^k. Provided k is small, we'll sometimes treat this as a primitive operation.

Data Structures

Data structures support algorithms, making them faster. Algorithms support data structures (as basic operations), making them more efficient. We'll look at algorithms on simple data structures (linear arrays) at first, then later we'll look at more sophisticated data structures and the algorithms that support them.

Lecture 4
A Tale of Two Algorithms

Insertion Sort and MergeSort in pseudocode. MergeSort uses the Divide-and-Conquer technique, with Merge as the combining subalgorithm.

Analysis: InsertionSort takes $\Theta(n^2)$ (quadratic) time to sort n numbers, in the worst case. In the best case, when the list is already sorted or almost sorted, the running time is $\Theta(n)$ (linear) time. The average case (over all possible arrangements of the initial data) is $\Theta(n^2)$, sort of because each item is inserted about half way into the sorted list to its left, on average.

Merge takes time $\Theta(m+n)$ to merge a sorted list of m items with a sorted list of n items. Since the total input size is $m+n$, this makes Merge linear time.

Since MergeSort is recursive, we cannot do a direct tally of the total running time like we did with InsertionSort. Instead, we let $T(n)$ be the (worst-case) time to MergeSort an array of n items. Then we do a direct tally of the time for each line of code, and we get the relation

$$T(n) = 2T(n/2) + \Theta(n).$$

This is an example of a *recurrence relation*. We'll see how to solve this and other recurrence relations soon.

Lecture 6
Recurrences

When an algorithm is recursive, we cannot just add up the component times directly, since we don't know how long the recursive calls take. Instead, if we let $T(n)$ be the (worst case) time of a given recursive algorithm A, then adding up the times gives us an equation expressing $T(n)$ in terms of $T(m)$ for $m < n$.

For example, suppose A takes a list of n numbers, and

- in the base case ($n = 1$, say) runs in constant time, and
- otherwise calls itself recursively twice, each time on a list of size $\lfloor n/2 \rfloor$, as well as taking $\Theta(n)$ time outside of the recursive calls.

Then we know that

$$\begin{aligned} T(1) &= O(1), \\ T(n) &= 2T(\lfloor n/2 \rfloor) + \Theta(n), \end{aligned}$$

where the second equation assumes $n > 1$. This is an example of a *recurrence relation*. It uniquely determines the asymptotic growth rate of $T(n)$. The time for Mergesort roughly satisfies this recurrence relation. More accurately, it satisfies

$$\begin{aligned} T(1) &= O(1), \\ T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n). \end{aligned}$$

To get a relation that uniquely defines values (not just the asymptotics) of a function bounding the run time, we supply constants in place of the asymptotic notation:

$$\begin{aligned} T(1) &= d, \\ T(n) &= 2T(\lfloor n/2 \rfloor) + cn, \end{aligned}$$

where c and d are positive constants. This uniquely determines the value of $T(n)$ for all positive integers n, and allows us to manipulate it more directly.

When we only care about asymptotics, we often omit c (tacitly assuming $c = 1$), as well as omitting the base case, which is always of the form, "if n is bounded by (some constant), then $T(n)$ is bounded by (some constant)." The actual choice of these constants does not affect the asymptotic growth rate of $T(n)$, so we can just ignore this case altogether.

Solving Recurrence Relations

We now concentrate on techniques to solve recurrence relations. There are three main approaches: the substitution method, the tree method, and the master method.

The Substitution Method

This method yields a rigorous proof by induction that a function given by a recurrence relation has a given asymptotic growth rate. It is, in some sense, the final word in the analysis of that function. A drawback is that you have to guess what the growth rate is before applying the method.

Consider the relation

$$\begin{aligned} T(0) &= 1, \\ T(n) &= 3T(\lfloor n/4 \rfloor) + n^2 \qquad (\text{for } n > 0), \end{aligned}$$

where $c > 0$ is a constant. Clearly $T(n) = \Omega(n^2)$, because $T(n) \geq n^2$ for all $n > 0$. We guess that this is tight, that is, $T(n) = O(n^2)$. Guessing the asymptotic growth rate is the first step of the method.

Now we try to prove our guess; we must show (by induction) that there is an n_0 and constant $c > 0$ such that

$$T(n) \leq cn^2 \tag{4}$$

for all $n \geq n_0$. For the inductive step, we assume that (4) holds for all values less than n, and show

that it therefore holds for n (where n is large enough). We have

$$\begin{aligned} T(n) &= 3T(\lfloor n/4 \rfloor) + n^2 \\ &\leq 3(c\lfloor n/4 \rfloor^2) + n^2 \qquad \text{(inductive hyp.)} \\ &\leq 3c(n/4)^2 + n^2 \\ &= \left(\frac{3}{16} \cdot c + 1\right) n^2. \end{aligned}$$

We're done with the inductive case if we can get the right-hand side to be $\leq cn^2$. This will be the case provided

$$\frac{3}{16} \cdot c + 1 \leq c,$$

or equivalently, $c \geq 16/13$. Thus, so far we are free to choose c to be anything at least $16/13$. The base case might impose a further constraint, though.

Now the base case, where we decide n_0. Note that $T(0) = 1 > c \cdot 0^2$ for *any* c, so we cannot choose $n_0 = 0$. We can choose $n_0 = 1$, though, as long as we establish (4) for enough initial values of n to get the induction started. We have

$$\begin{aligned} T(1) &= 4, \\ T(2) &= 7, \\ T(3) &= 12. \end{aligned}$$

All higher values of n reduce inductively to these three, so it suffices to pick $n_0 = 1$ and $c = 4$, so that (4) holds for $n \in \{1, 2, 3\}$. Since $4 \geq 16/13$, our choice of c is consistent with the constraint we established earlier. By this we have shown that $T(n) = O(n^2)$, and hence $T(n) = \Theta(n^2)$.

In the future, we will largely ignore floors and ceilings. This almost never gets us into serious trouble, but doing so make our arguments less than rigorous. Thus after analyzing a recurrence by ignoring the floors and ceilings, if we want a rigorous proof, we should use our result as the guess in an application of the substitution method on the original recurrence.

Changing Variables

Sometimes you can convert an unfamiliar recurrence relation into a familiar one by changing variables. Consider

$$T(n) = 2T(\sqrt{n}) + \lg n. \tag{5}$$

Let S be the function defined by $S(m) = T(2^m)$ for all m. Then from (5) we get

$$S(m) = T(2^m) = 2T(2^{m/2}) + \lg 2^m = 2S(m/2) + m. \tag{6}$$

So we get a recurrence on S without the square root. This is essentially the same recurrence as that for MergeSort. By the tree or master methods (later), we can solve this recurrence for S to get $S(m) = \Theta(m \lg m)$. Thus,

$$T(n) = S(\lg n) = \Theta(\lg n \lg \lg n).$$

The Tree Method

The execution of a recursive algorithm can be visualized as a tree. Each node of the tree corresponds to a call to the algorithm: the root being the initial call, and the children of a node

being the recursive calls made at that node. Each node is labeled with the time taken in that node. We can often solve a recurrence by looking at the structure of the corresponding tree.

Consider the recurrence

$$T(n) = T(n/3) + T(2n/3) + n.$$

The root is labeled n, with two children, the left labeled $n/3$ and the right labeled $2n/3$, etc. The shallowest *leaf* in the tree (that is, when the argument is about 1) is along the leftmost path, at depth $\log_3 n$. The deepest leaf is along the rightmost path, at depth $\log_{3/2} n$. Now we add up the labels level-by-level, starting at the root (level 0). To get a lower bound on $T(n)$, we only count full levels, so we go down to the level of the shallowest leaf. Each of these levels is found to add up to n, so the first $\log_3 n + 1$ levels give a total of $n(\log_3 n + 1)$. Thus $T(n) = \Omega(n \lg n)$. For an upper bound, we continue counting all the levels in the tree as if they were full (and thus added up to n). We thus get an upper bound of $n(\log_{3/2} n + 1) = O(n \lg n)$. Thus $T(n) = \Theta(n \lg n)$.

Lecture 7
Recurrences (continued)

The Master Method

The master method solves recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

for a wide variety of functions $f(n)$ and values of a and b. It derives directly from the following theorem:

Theorem 7 (Master Theorem) *Let $f(n)$ be a function, and let a and b be real with $a \geq 1$ and $b > 1$. Suppose $T(n)$ is given by the recurrence*

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to be either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then, letting $t = \log_b a \geq 0$,

1. *if there is an $\varepsilon > 0$ such that $f(n) = O(n^{t-\varepsilon})$, then $T(n) = \Theta(n^t)$.*
2. *if $f(n) = \Theta(n^t)$, then $T(n) = \Theta(n^t \lg n)$.*
3. *if there is an $\varepsilon > 0$ such that $f(n) = \Omega(n^{t+\varepsilon})$, and there is a $c < 1$ such that $af(n/b) \leq cf(n)$ for all sufficiently large n, then $T(n) = \Theta(f(n))$.*

Proof We will prove the theorem without worrying about floors and ceilings. Also, we'll only prove the most important special cases of the theorem—those in which $f(n) = n^d$ for some real $d \geq 0$. The general theorem (including floors/ceilings) can be reduced rather easily to these cases.

Let $f(n) = n^d$ for $d \geq 0$. If $d < t$, then we are in Case 1. If $d = t$, then Case 2, otherwise if $d > t$ then it is Case 3. In any case, using the tree method, we get a completely balanced tree of

depth $\log_b n$, where the total for level i is $a^i f(n/b^i)$. Thus,

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n} a^i f(n/b^i) \\ &= \sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^d \\ &= n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i \\ &= n^d \sum_{i=0}^{\log_b n} r^i, \end{aligned}$$

where $r = a/b^d$. This is clearly a finite geometric series. Noting that $b^d = a/r$, and thus $\log_b r = \log_b a - d = t - d$, we see that $r > 1$, $r = 1$, or $r < 1$ just as $t > d$, $t = d$, or $t < d$, respectively.

We consider the three possibilities:

$t > d$. This corresponds to Case 1 of the theorem. We have,

$$\begin{aligned} T(n) &= n^d \left(\frac{r^{\log_b n+1} - 1}{r - 1}\right) \\ &= \frac{1}{r-1} n^d (r \cdot r^{\log_b n} - 1) \\ &= \frac{r}{r-1} n^d \left(r^{\log_b n} - \frac{1}{r}\right) \\ &= \frac{r}{r-1} n^d \left(n^{\log_b r} - \frac{1}{r}\right) \\ &= \frac{r}{r-1} n^d \left(n^{t-d} - \frac{1}{r}\right) \\ &= \frac{r}{r-1} \left(n^t - \frac{n^d}{r}\right) \\ &= \Theta(n^t) \end{aligned}$$

as in Case 1. Note that $n^d = o(n^t)$.

$t < d$. This corresponds to Case 3 of the theorem. We first verify the regularity property of $f(n) = n^d$:

$$af(n/b) = (a/b^d)n^d = r \cdot n^d = r \cdot f(n),$$

and thus we can take $c = r < 1$ to make the regularity property hold. Next, since $0 < r < 1$,

we see that for all $n \geq 1$,

$$\begin{aligned} n^d &= n^d \sum_{i=0}^{0} r^i \\ &\leq n^d \sum_{i=0}^{\log_b n} r^i \\ &= T(n) \\ &\leq n^d \sum_{i=0}^{\infty} r^i \\ &= \frac{n^d}{1-r}, \end{aligned}$$

and thus $T(n) = \Theta(n^d) = \Theta(f(n))$, as in Case 3.

$t = d$**.** This corresponds to Case 2 of the theorem. We have $r = 1$, so

$$T(n) = n^d \sum_{i=0}^{\log_b n} 1 = n^d(\log_b n + 1) = \Theta(n^d \lg n),$$

as in Case 3.

□

Remark. In Case 1 of the theorem, the tree is bottom-heavy, that is, the total time is dominated by the time spent at the leaves. In Case 3, the tree is top-heavy, and the time spent at the root dominates the total time. Case 2 is in between these two. In this case, the tree is "rectangular," that is, each level gives about the same total time. Thus the total time is the time spent at level 0 (the root, i.e., $\Theta(n^t)$) times the number of levels (i.e., $\Theta(\lg n)$).

The Master Method

Use the master theorem to solve the following recurrences:

1. $T(n) = 2T(n/2) + n$ (Case 2),
2. $T(n) = 8T(n/3) + n^2$ (Case 3),
3. $T(n) = 9T(n/3) + n^2$ (Case 2),
4. $T(n) = 10T(n/3) + n^2$ (Case 1),
5. $T(n) = 2T(2n/3) + n^2$ (Case 3),
6. $T(n) = T(9n/10) + 1$ (Case 2).

The first recurrence is that of MergeSort. The Master Theorem gives a time of $\Theta(n \lg n)$. In each case, go through the following steps:

1. What is a?
2. What is b?

3. Compare $\log_b a$ with the exponent of $f(n)$.

4. Determine which case of the Master Theorem applies (if any)

5. Read off the asymptotic expression for $T(n)$ from the Theorem.

Fast Integer Multiplication

To see a good example of divide-and-conquer with an analysis using the Master method, we look at the task of multiplying (large) integers. Let's assume we have two integers $b, c \geq 0$, represented in binary, with n bits each. Here, n is assumed to be large, so we cannot assume as we usually do that b and c can be added, subtracted, or multiplied in constant time.

We imagine that the b and c are both represented using arrays of n bits: $b = b_{n-1} \cdots b_0$ and $c = c_{n-1} \cdots c_0$, where the b_i and c_i are individual bits (leading 0s are allowed). Thus,

$$\begin{aligned} b &= \sum_{i=0}^{n-1} b_i 2^i, \\ c &= \sum_{i=0}^{n-1} c_i 2^i. \end{aligned}$$

Addition

The usual sequential binary add-with-carry algorithm that we all learned in school takes time $\Theta(n)$, since we spend a constant amount of time at each column, from right to left. The sum is representable by $n+1$ bits (at most). This algorithm is clearly asymptotically optimal, since the produce the correct sum we must at least examine each bit of b and of c.

Subtraction

Similar to addition, the usual subtract-and-borrow algorithm takes $\Theta(n)$ time, which is clearly asymptotically optimal. The result can be represented by at most n bits.

Multiplication

Now the interesting bit. If we multiply b and c using the naive grade school algorithm, then it takes quadratic ($\Theta(n^2)$) time. Essentially, this algorithm is tantamount to expanding the product bc according to the expressions above:

$$bc = \left(\sum_{i=0}^{n-1} b_i 2^i \right) \left(\sum_{j=0}^{n-1} c_j 2^j \right) = \sum_{i,j} b_i c_j 2^{i+j},$$

then adding everything up term by term. There are n^2 many terms.

Multiplying with Divide-and-Conquer

If $n = 1$, then the multiplication is trivial, so assume that $n > 1$. Let's further assume for simplicity n is even. In fact, we can assume that n is a power of 2. If not, pad each number with leading 0s to the next power of 2; this doubles the input size at worst.

Let $m = n/2$. Split b and c up into their m least and m most significant bits. Let b_ℓ and b_h be the numbers given by the low m bits and the high m bits of b, respectively. Similarly, let c_ℓ and c_h be the low and high halves of c. Thus, $0 \leq b_\ell, b_h, c_\ell, c_h < 2^m$ and

$$\begin{aligned} b &= b_\ell + b_h 2^m, \\ c &= c_\ell + c_h 2^m. \end{aligned}$$

We then have

$$bc = (b_\ell + b_h 2^m)(c_\ell + c_h 2^m) = b_\ell c_\ell + (b_\ell c_h + b_h c_\ell)2^m + b_h c_h 2^n.$$

This suggests that we can compute bc with four recursive multiplications of pairs of m-bit numbers—$b_\ell c_\ell$, $b_\ell c_h$, $b_h c_\ell$, and $b_h c_h$—as well as $\Theta(n)$ time spent doing other things, namely, some additions and multiplications by powers of two (the latter amounts to arithmetic shift of the bits, which can be done in linear time.) The time for this divide-and-conquer multiplication algorithm thus satisfies the recurrence

$$T(n) = 4T(m) + \Theta(n) = 4T(n/2) + \Theta(n).$$

The Master Theorem (Case 1) then gives $T(n) = \Theta(n^2)$, which is asymptotically no better than the naive algorithm.

Can we do better? Yes. Split b and c up into their low and high halves as above, but then recursively compute only three products:

$$\begin{aligned} x &:= b_\ell c_\ell, \\ y &:= b_h c_h, \\ z &:= (b_\ell + b_h)(c_\ell + c_h). \end{aligned}$$

Now you should verify for yourself that

$$bc = x + (z - y - x)2^m + y2^n,$$

which the algorithm then computes. How much time does this take? Besides the recursive calls, there's a linear time's worth of overhead: additions, subtractions, and arithmetic shifts. There are three recursive calls—computing x, y, and z. The numbers x and y are products of two m-bit integers each, and z is the product of (at most) two $(m+1)$-bit integers. Thus the running time satisfies

$$T(n) = 2T(n/2) + T(n/2 + 1) + \Theta(n).$$

It can be shown that the "+1" doesn't affect the result, so the recurrence is effectively

$$T(n) = 3T(n/2) + \Theta(n),$$

which yields $T(n) = \Theta(n^{\lg 3})$ by the Master Theorem.[1] Since $\lg 3 \doteq 1.585 < 2$, the new approach runs significantly faster asymptotically.

This approach dates back at least to Gauss, who discovered (using the same trick) that multiplying two complex numbers together could be done with only three real multiplications instead of the more naive four. The same idea has been applied to matrix multiplication by Volker Strassen.

Lecture 8
Sorting and Order Statistics: Heapsort

[1] If you're really worried about the "+1,", you should verify that $T(n) = \Theta(n^{\lg 3})$ directly using the substitution method. Alternatively, you can modify the algorithm a bit so that only m-bit numbers are multiplied recursively and the overhead is still $\Theta(n)$.

Finally, we look at algorithms for some specific problems. Note that we are skipping Chapter 5, although we will use some of its techniques eventually.

The Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Items usually stored in an array (or a linked list), perhaps as keys of records with additional satellite data. If the amount of satellite data in each record is large, we usually only include a pointer to it with the key, in order to minimize data movement while sorting. These are programming details rather than algorithmic—i.e., methodological—issues, so we usually consider the array to consist of numbers only.

Sorting is fundamental in the study of algorithms, with a wide variety of applications.

We've seen Insertion Sort ($\Theta(n^2)$ time in the worst case, but sorts in place) and MergeSort ($\Theta(n \lg n)$ time), but the Merge subroutine does not rearrange data in place.

We'll see two other in-place sorting routines: HeapSort ($\Theta(n \lg n)$ worst-case time) and QuickSort ($\Theta(n^2)$ worst-case time, but $\Theta(n \lg n)$ average-case time). Both sort in place ($O(1)$ extra space used).

The Order Statistics Problem

The ith order statistic of a (multi)set of n numbers is the ith smallest number in the set. We can find the ith order statistic by sorting the numbers in the set, then indexing the ith element. This takes time asymptotically equal to that of sorting, e.g., $\Theta(n \lg n)$ for MergeSort. But there is actually a linear time algorithm for finding the ith order statistic.

Heaps

A heap is used to implement a priority queue, and is also used in Heapsort

Heaps are collections of objects with keys taken from a totally ordered universe. There are two types of heap: min heap and max heap. Min heaps support the following operations:

Insertion Adding an element to the heap

Deletion Removing an element with minimum key value

Finding the minimum Returing an object with minimum key value (the heap is unchanged)

Decrease key Decreasing the key value of an element of the heap

Max heaps support the same operations with "minimum" replaced with "maximum" and "decrease" replaced with "increase."

An Implementation

Binary Heap: array A with two attributes: *length*$[A]$ and *heap-size*$[A]$, which is at most *length*$[A]$. The elements of the heap reside in $A[1 \ldots \textit{heap-size}[A]]$ and can be pictured as an almost full binary tree. (A binary tree is *almost full* if all its leaves are either on the same level or on two adjacent levels with the deeper leaves as far left as possible.

The correspondance between node locations on the tree and array indices is obtained by traversing the tree in level order, left to right within each level. The root is at index 1. The left and right children of the node with index i have indices $2i$ and $2i + 1$, respectively, and for any nonroot note

with index $i > 1$, the index of its parent is $\lfloor i/2 \rfloor$. So, for example, the rightmost leaf on the bottom level has index *heap-size*$[A]$. Thus we have the following functions running in $O(1)$ time:

Parent(i)
    return $\lfloor i/2 \rfloor$

Left(i)
    return $2i$

Right(i)
    return $2i + 1$

The elements of the heap are kept in either *max heap order* or *min heap order*, depending on the type of heap. Max heap order means that for any $2 \leq i \leq$ *heap-size*$[A]$ we have $A[$*parent*$(i)] \geq A[i]$ (the inequality is reversed for min heaps).

Lecture 9
Heaps and Heapsort(continued)

Recall:
$A[1 \ldots$ *length*$[A]]$ holds heap items (numbers)
The actual heap is kept in $A[1 \ldots$ *heap-size*$[A]]$, where *heap-size*$[A]$ is always less than *length*$[A]$.
Now, we show how to take an arbitrary array $A[1 \ldots n]$ and put it into max-heap order. This uses MaxHeapify, an important subroutine for other heap operations.

The idea is that we make each subtree a max-heap, starting from the bottom of the tree and working up toward the root. This means that when we make the tree rooted at index i into a heap, we can assume that the subtrees rooted at Left(i) and Right(i) are heaps already.

Each leaf is already a max-heap, so we start with the nonleaf with highest index, that is, Parent(*heap-size*$[A]$).

```
BuildMaxHeap(A, n)
// Makes a max-heap out of the
// first n elements of A.
// Precondition: 1 ≤ n ≤ length[A]
    for i ←Parent(n) downto 1 do
        MaxHeapify(A, n, i)
    heap-size[A]←n
```

```
MaxHeapify(A, n, i)
// Makes the subtree of A[1 . . . n]
// rooted at i into a max-heap
// Precondition: 1 ≤ i ≤ n
// Precondition: the subtrees rooted at
// Left(i) and Right(i) are
// in max-heap order
// Method: cascade the root down into the tree
```

```
m←j←i
repeat forever
    if Left(j) ≤ n and A[Left(j)] > A[m] then
        m←Left(j)
    if Right(j) ≤ n and A[Right(j)] > A[m] then
        m←Right(j)
    // m is index of largest item
    // among A[j], A[left(j)], A[right(j)].
    if m = j then
        return
    else
        swap A[j] ↔ A[m]
        j←m // this doubles j (at least)
```

We need to show that these algorithms are correct. We leave that for later, but now just mention that if MaxHeapify correctly does what it is supposed to, then clearly MakeMaxHeap is correct as well.

We now look at the run time of MaxHeapify and MakeMaxHeap. In MaxHeapify, each iteration of the repeat-loop takes $O(1)$ time, as well as all the stuff outside the repeat-loop. Thus the time for MaxHeapify(A, n, i) is asymptotically equal to the number of iterations of the repeat-loop. This is clearly equal to one plus the depth of the subtree rooted at index i. Clearly, a longest path from the root in this tree is down the left spine, and the length of this path is the largest k such that k applications of Right to i is $\leq n$. That is, the largest k such that $i2^k \leq n$. Thus $k = \lfloor \lg(n/i) \rfloor$, and the number of iterations of the repeat-loop is thus $1 + \lfloor \lg(n/i) \rfloor$. Therefore, the time for MaxHeapify(A, n, i) is $\Theta(\lfloor \lg(n/i) \rfloor) = \Theta(\lg(n/i))$.

To get the total time $T(n)$ for MakeMaxHeap(A, n), first we note that clearly, $T(n) = \Omega(n)$, since it has at least $\lfloor n/2 \rfloor$ loop iterations. To get an upper bound, we take a sum and split it. Let m be least such that $n < 2^m$. Note that $2^m \leq 2n$. By the analysis of MaxHeapify, we get that

$$T(n) = \Theta\left(\sum_{i=1}^{\lfloor n/2 \rfloor} \lfloor lg(n/i) \rfloor\right).$$

So for large enough n, there is a constant C such that

$$\begin{aligned}
T(n) &\leq C \sum_{i=1}^{\lfloor n/2 \rfloor} \lfloor \lg(n/i) \rfloor \\
&\leq C \sum_{i=1}^{2^m-1} \lfloor \lg(2^m/i) \rfloor \\
&= C \sum_{i=1}^{2^m-1} \lfloor m - \lg i \rfloor \\
&= C \sum_{i=1}^{2^m-1} (m - \lceil \lg i \rceil) \\
&= C \sum_{k=0}^{m-1} \sum_{j=0}^{2^k-1} (m - \lceil \lg(2^k + j) \rceil) \\
&\leq C \sum_k \sum_j (m-k) \\
&= C \sum_k (m-k) 2^k \\
&= C \sum_{r=1}^{m} r 2^{m-r} \\
&= C 2^m \sum_{r=1}^{m} r/2^r \\
&\leq C 2^m \sum_{r=1}^{\infty} r/2^r.
\end{aligned}$$

We know that this last sum converges to some constant D (exercise: what is D?). Thus for all sufficiently large n,

$$T(n) \leq CD2^m \leq 2CDn = O(n).$$

Thus MakeMaxHeap(A, n) takes time $\Theta(n)$.

We now implement other heap operations. We do this for max-heaps.

FindMax(A)
// Precondition: *heap-size*$[A] \geq 1$
return $A[1]$

DeleteMax(A) // Precondition: *heap-size*$[A] \geq 1$
$A[1] \leftarrow A[$*heap-size*$[A]]$
heap-size$[A] \leftarrow$*heap-size*$[A] - 1$
MaxHeapify(A, *heap-size*$[A], 1$)

Finally, HeapSort:

```
HeapSort(A, n)
// Sort A[1...n]
// Precondition: n ≤ length[A]
    MakeMaxHeap(A, n)
    while heap-size[A] > 0 do
        save ←FindMax(A)
        DeleteMax(A)      // also decrements heap-size[A]
        A[heap-size[A] + 1] ←save
```

As we've seen, MakeMaxHeap(A, n) takes $\Theta(n)$ time. The while-loop iterates n times, and the bulk of the work of each iteration is done by DeleteMax. We've seen that DeleteMax called with a heap of size i takes time $\Theta(\lg i)$ in the worst case, so the total time taken by the calls to DeleteMax in the while-loop is asymptotically equal to

$$D(n) = \sum_{i=1}^{n} \lg i.$$

The fact that $D(n) = \Theta(n \lg n)$ follows from the following inequalities, which hold for $n > 0$:

$$\begin{aligned}
\frac{n(\lg n - 1)}{2} &= \frac{n}{2} \lg \frac{n}{2} \\
&\leq \sum_{i=\lceil n/2 \rceil}^{n} \lg \frac{n}{2} \\
&\leq \sum_{i=\lceil n/2 \rceil}^{n} \lg i \\
&\leq D(n) \\
&\leq \sum_{i=1}^{n} \lg n \\
&= n \lg n.
\end{aligned}$$

Lecture 10
Quicksort

Quicksort is a divide-and-conquer sort that takes $\Theta(n^2)$ time in the worst case but $\Theta(n \lg n)$ time in the average case. Like Mergesort, it splits the list in two, sorts both sublists recursively, then recombines the results. Unlike Mergesort, the major work is in splitting the list, while the recombination is trivial.

Quicksort uses a Partition subroutine to split the lists. Partition(A, p, r) takes a list $A[p \ldots r]$ (assuming $p \leq r$) and rearranges its elements so that there is an index q such that $p \leq q \leq r$ and $A[i] \leq A[q] \leq A[j]$ for all i, j such that $p \leq i < q < j \leq r$. The index q is returned by the Partition procedure. The value $A[q]$ is called the *pivot*. Once Partition returns q, we simply sort the sublists $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ recursively. When the recursive sorting is done, it is clear that the list is entirely sorted, so there is no recombining step needed.

```
Quicksort(A, p, r)
// Sorts A[p ... r]
    if p < r then
        q ←Partition(A, p, r)
        Quicksort(A, p, q − 1)
        Quicksort(A, q + 1, r)
```

Quicksort runs fastest when the list is split into roughly equal-sized sublists each time. Thus the Partition procedure should choose the partition to be at least reasonably close to the median. If we expect randomly arranged initial data, then any element of $A[p \ldots r]$ is just as likely to be near the median as any other, so we could naively just choose $A[p]$ as the pivot. Let's call this value x.

The partitioning procedure described here is closer to Hoare's original partitioning procedure than the one given in the textbook. It is not particularly better or worse than the textbook's, just different. We start from the extremes of the list and work towards the middle, whenever we come upon a pair of indices $i < j$ for which $A[i] > x > A[j]$, we swap $A[i]$ with $A[j]$ and continue. We end up somewhere in the array, and this index is the return value q. We move x into $A[q]$ and return.

Verifying the correctness of this procedure is a tricky business, so the pseudocode includes several assertions to help with this. You should verify that all assertions are true each time they occur.

```
Partition(A, p, r)
// Precondition: p ≤ r
    x←A[p]      // x is the pivot
    i←p
    j←r + 1
    // A[p] = x throughout the following loop.
    repeat
        // i < j and A[i] ≤ x.
        repeat j←j − 1 until j ≤ i or A[j] < x
        // j ≥ i and A[j] ≤ x.
        // All elements of A[j + 1 ... r] are ≥ x.
        repeat i←i + 1 until i ≥ j or A[i] > x
        // All elements of A[p ... i − 1] are ≤ x.
        if i < j then
            // A[i] > x > A[j]
            swap(A[i], A[j])
    until i ≥ j
    // p ≤ j ≤ r and j ≤ i.
    // (Actually, either i = j or i = j + 1.)
    // A[j] ≤ x.
    // All elements of A[j + 1 ... r] are ≥ x.
    // All elements of A[p ... i − 1] are ≤ x.
    // Thus all elements of A[p ... j] are ≤ x.
    swap(A[p], A[j])
```

// $A[j] = x$ and $A[p] \leq x$.
// Thus all elements of $A[p \ldots j]$ are $\leq x$.
$q \leftarrow j$
// $A[q] = x$.
return q

The time taken by Partition is clearly asymptotically dominated by the number of times i is incremented and j is decremented in the inner repeat-loops, since each happens at least once in any iteration of the outer repeat-loop. But $i = p$ and $j = r + 1$ initially, and afterwards, $j \leq i \leq j + 1$, so the total number of increment/decrement operations is between $r - p + 1$ and $r - p + 2$. Letting $n = r - p + 1$ be the number of elements in the list, we see that Partition takes time linear in n, i.e., $\Theta(n)$.

In this version of Partition, the pivot is always the first element of the (sub)list. If the initial n-element list is already sorted, then the first element is also the least, so the partition is into an empty list and a list of size $n - 1$, which is also sorted. Thus if $T(n)$ is the time for Quicksort in this case, we have

$$T(n) = T(n - 1) + \Theta(n),$$

which is easily solved to get $T(n) = \Theta(n^2)$. This is the worst case behavior of Quicksort. The best case is when the pivot is always close to the median. Then we get essentially the same recurrence as with Mergesort, yielding $T(n) = \Theta(n \lg n)$. Assuming a uniformly random input distribution, one can also show that the average time for Quicksort over all possible input arrangements is also $\Theta(n \lg n)$; the pivot is close enough to the median most of the time on average. (This analysis is done in the textbook.)

Unfortunately, it is often not reasonable to assume a completely random distribution. For example, the list may be almost sorted, which leads in our case behavior close to the worst case. Some alternative deterministic partition strategies can help with some of these cases, for example, choosing the pivot to be the median of $A[p]$, $A[r]$, and $A[\lfloor p + r/2 \rfloor]$ (the so-called median-of-three strategy), which gives good behavior when the list is almost sorted. Still, there are initial arrangments (rather contrived) that make the median-of-3 strategy exhibit worst-case behavior.

Another approach is to choose the pivot *at random* from $A[p \ldots r]$. This procedure is no longer deterministic, but *randomized*. We assume a procedure Random(p, r) that returns an integer in the range $p \ldots r$ *uniformly* at random, i.e., so that each integer in the range has an equal probability of being returned. We also assume that successive calls to Random return *independently* random—i.e., uncorrelated—values. Finally, it is reasonable to assume that Random runs in constant time for "reasonable" values of p and q.

RandomPartition(A, p, r)
$m \leftarrow$Random(p, r)
swap$(A[p], A[m])$
return Partition(A, p, r)

When analyzing a randomized algorithm, we usually do not average over different inputs, but rather over the random choices made in the algorithm for any fixed input. We now see that for any initial arrangement of $A[p \ldots r]$ the expected running time of RandomizedQuicksort(A, p, r) (averaged over all possible sequences of return values of Random) is $\Theta(n \ln n)$, where $n = r -$

$p+1$ and RandomizedQuicksort is the same as Quicksort except that we replace Partition with RandomPartition.

Consider the first call to RandomizedQuicksort(A, p, r). Since the pivot is chosen uniformly at random from $A[p \ldots r]$, the return value of the first call to RandomPartition(A, p, r) is distributed uniformly at random in the range $p \ldots r$. Thus if k is the size of the left-hand list (the size of the right-hand list then being $n-k-1$), its possible values range from 0 to $n-1$, inclusive, with each size occurring with probability $1/n$. Let $E(n)$ be the expected time for RandomizedQuicksort of n elements. Then we see that $E(n)$ is an average over the expected running times for each possible value of k. Thus for all $n \geq 2$ (that is, when partition occurs and recursive calls are made),

$$\begin{aligned} E(n) &= \Theta(n) + \frac{1}{n}\sum_{k=0}^{n-1}(E(k) + E(n-k-1)) \\ &= \Theta(n) + \frac{2}{n}\sum_{k=0}^{n-1} E(k) \\ &\leq an + \frac{2}{n}\sum_{k=0}^{n-1} E(k), \end{aligned}$$

where a is some constant such that an bounds the running time of the initial call to RandomPartition.

We get an upper bound for $E(n)$ using the substitution method. We guess that $E(n) \leq cn\lg n + b$ for some constants $b, c > 0$, since we guess (we hope!) that the average case should act like $\Theta(n \lg n)$. (The additive term b is important for the base case. Since $\lg 1 = 0$ and $0 \lg 0 = 0$ by convention, the equation above cannot hold for $n = 0$ or $n = 1$ unless b is big enough, i.e., $b \geq \max(E(0), E(1))$.)

So we fix $n \geq 2$ and assume that $E(m) \leq cm \ln m + b$ for all $m < n$, where $\ln n$ is the natural logarithm (to base e) of n. We wish to show that $E(n) \leq cn \ln n + b$. We use the natural logarithm here because we will use an integral approximation. We have

$$\begin{aligned} E(n) &\leq an + \frac{2}{n}\sum_{k=0}^{n-1} E(k) \\ &\leq an + \frac{2}{n}\sum_{k=0}^{n-1}(ck \ln k + b) \\ &= an + 2b + \frac{2c}{n}\sum_{k=1}^{n-1} k \ln k \\ &\leq an + 2b + \frac{2c}{n}\int_1^n x \ln x \, dx \\ &= an + 2b + \frac{2c}{n}\left(\frac{n^2 \ln n}{2} - \frac{n^2}{4} + \frac{1}{4}\right) \\ &= cn \ln n - c\left(\frac{n^2 - 1}{2n}\right) + an + 2b \\ &\leq cn \ln n - \frac{c(n-1)}{2} + an + 2b \\ &\leq cn \ln n + b, \end{aligned}$$

with the last inequality holding provided

$$c \geq \frac{2(an+b)}{n-1}.$$

Since $n \geq 2$, we have

$$\frac{2(an+b)}{n-1} \leq \frac{4(an+b)}{n} \leq 4a+2b,$$

and therefore it suffices first to let

$$b = \max(E(0), E(1))$$

to handle the case where $n < 2$, then to let

$$c = 4a+2b.$$

This proves that $E(n) = O(n \ln n) = O(n \lg n)$ as desired.

Putting it all together, we see that RandomizedQuicksort runs in expected time $\Theta(n \lg n)$.

Lecture 11
Comparison-Based Sorting Lower Bounds, Selection Problem

A *comparison-based* sorting algorithm is one which only inspects the data when comparing two elements. There are other sorting algorithms that take advantage of certain properties of the data, such as: the data are integers in a certain range, or each datum is given in binary, for example. These are not comparison-based, since they look at and manipulate the data in ways other than simple comparisons.

All the sorting algorithms we've seen so far (Insertion Sort, MergeSort, HeapSort, QuickSort) are comparison-based.

Today we show that any deterministic (e.g., not randomized) comparison-based sorting algorithm requires $\Omega(n \lg n)$ time to sort n items in the worst case. In particular, in the worst case, such an algorithm makes $\Omega(n \ln n)$ comparisons. This shows that MergeSort and HeapSort are asymptotically optimal worst-case comparison-based sorting algorithms. One can also show (we won't) that any deterministic or randomized comparison-based sorting algorithm must also take $\Omega(n \lg n)$ time in the average case, so QuickSort is optimal in the average case.

For the proof, we model the behavior of a comparison-based algorithm by a *decision tree*. Each nonleaf node of our decision tree is labeled with a comparison, and its subtrees represent the alternative computations based on the answer to the comparison. Our comparisons are binary (e.g., "Is $A[17] < A[14]$?"), so our decision tree is a binary tree. A particular execution of the algorithm corresponds to a path through the tree from the root to a leaf. Each intermediate node along the path corresponds to the algorithm making the corresponding comparison, and the path continues according to the answer to the comparison. The leaf represents the end of the computation.

Let P be any deterministic comparison-based algorithm that correctly sorts any initial arrangement of n distinct items. Let T be its corresponding decision tree.

Claim 8 *T has at least $n!$ leaves.*

Proof There are $n!$ ways of arranging n distinct items in the input array. Each path through T effects a permutation of the items in the array (to get it sorted). Since any two different initial arrangements of the array must be permuted differently to sort them, the algorithm cannot take the same path through the tree on both initial arrangements (otherwise that would lead to the same permutation of the two different arrangements, so at least one of them would not be sorted in the end). Thus by the pigeon hole principle, the number of distinct root-leaf paths in the tree—and hence the number of leaves—must be at least the number of possible initial arrangements of the data, that is, $n!$. □

Exactly $n!$ of the leaves of T are actually reachable by the algorithm on some input arrangement. If T has more than $n!$ leaves, then some are not reachable. We'll assume that T is pruned of all its unreachable nodes.

Lemma 9 *Any binary tree with k leaves has depth at least $\lg k$.*

Proof Suppose T is a binary tree of depth d that has the maximum number of leaves possible of any tree of depth at most d. Then all the leaves of T are on level d: if there were a leaf on some level $i < d$, then we could replace that leaf with a full binary subtree of depth $d - i$ with a net gain of $2^{d-i} - 1 > 0$ leaves. There are 2^d many nodes possible on level d, so T has 2^d leaves. Thus for *any* binary tree of depth d with k leaves, we must have

$$k \leq 2^d.$$

Taking the lg of both sides proves the lemma. □

Let T again be the decision tree for P acting on a list of n elements, where T is pruned of unreachable nodes. By the Claim, above, T has $n!$ many leaves, and hence by the Lemma, T has depth at least $\lg n!$, so there is some initial arrangement of the input that forces P to make at least $\lg n!$ comparisons. We'll be done if we can show that $\lg n! = \Omega(n \lg n)$.

The usual tricks will work here. We have, for large enough n,

$$\begin{aligned}
\lg n! &= \sum_{i=1}^{n} \lg i \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \lg i \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \lg(n/2) \\
&\geq (n/2) \lg(n/2) \\
&= (n/2)(\lg n - 1) \\
&\geq (n/4) \lg n \\
&= \Omega(n \lg n).
\end{aligned}$$

Order Statistics and Selection

Given an array A of n numbers, the *ith order statistic* of A is the ith smallest element of A, for $1 \leq i \leq n$. The*selection problem* is: given A and integer i, return the ith order statistic of A.

Instances of this problem include finding the minimum, maximum, median, 75th percentile, etc. in A.

Any of these selection instances require linear time at least, because the ith order statistic, for any i, cannot be determined without examining all the elements of the array. Conversely, finding the minimum and maximum both easily take linear time. How about the median? This looks harder. We can find the median in time $O(n \lg n)$, say, by first HeapSorting or MergeSorting the list, then picking out the middle element.

Can we do better? Yes. First, we'll give a randomized selection algorithm that runs in linear expected time (and quadratic worst-case time). This algorithm works well in practice.

Finally, we'll give a deterministic algorithm for selection that runs in worst-case linear time, and is thus asymptotically optimal. The deterministic algorithm, however, usually does worse than the randomized one, and so it is more of theoretical than practical interest.

Both algorithms use the same Partition subroutine as Quicksort. To find the ith order statistic in $A[p \ldots r]$ with $p < r$, the idea is to partition the array by some pivot value in the array. After partitioning, if the left list has length exactly $i - 1$, then the ith smallest element is equal to the pivot, so we just return it. Otherwise, if the left list has length at least i, then the ith order statistic is in the left list, so we recurse on the left list. Otherwise, we recurse on the right list.

The two algorithms only differ by how the partitioning is done.

Randomized Selection

The first algorithm uses RandomPartition.

```
RandomizedSelect(A, p, r, i)
// Returns the ith smallest element of A[p...r].
// Preconditions: p ≤ r and 1 ≤ i ≤ r − p + 1.
    if p = r then
        return A[p]
    // p < r
    q ←RandomPartition(A, p, r)
    // the left list is A[p...q − 1], and
    // the right list is A[q + 1...r].
    if q − p = i − 1 then
        // Return the pivot
        return A[q]
    if q − p > i − 1 then
        // Recurse on the left list
        return RandomizedSelect(A, p, q − 1, i)
    else
        // Recurse on the right list
        return RandomizedSelect(A, q + 1, r, i − (q − p + 1))
```

In the last line, we subtract $q - p + 1$ from i in the recursive call to account for the pivot and all the elements of the left list.

Let $E(n)$ be the expected running time for RandomizedSelect. Since RandomPartition (which takes linear time) selects a pivot uniformly at random from the list, the probability that the left list will have length k is the same for all $0 \leq k \leq n - 1$, namely, $1/n$. It is a reasonable assumption

that $E(n)$ is monotone increasing in n, which means that it will always take at least as long if we recurse on the larger of the two lists each time. Thus,

$$E(n) = \Theta(n) + \frac{1}{n}\sum_{k=0}^{n-1} \max(E(k), E(n-k-1)),$$

similar to the analysis of RandomizedQuicksort. The right-hand side is at most

$$\Theta(n) + \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} E(k).$$

One can then prove using the substitution method that $E(n) = O(n)$.

Lecture 12
Selection in Deterministic Linear Time

Defining the Median

For odd n, the median for an array of n elements (assume that they are all distinct) is defined to be the $(n+1)/2$-order statistic. If n is even, then the array actually has two medians, the $(n/2)$- and the $(n/2+1)$-order statistic. In statistics we usually define "the median" in this case to be the average of the two medians. Here, we will simply take the lower of the two. So for us, the median will be the $\lceil n/2 \rceil$-order statistic, regardless of the parity of n.

Selection in Deterministic Linear Time

For the deterministic linear time selection algorithm, the idea is that we choose a pivot that is guaranteed to be close enough to the median each time. This choice requires some work, and there is a delicate balance between how hard we are willing to work to find a good pivot versus the time saved by choosing a good pivot.

A Recurrence

Before getting to the algorithm, we will first solve an important recurrence.

Lemma 10 *Suppose α and β are real numbers with $0 < \alpha, \beta$ and $\alpha + \beta < 1$. Then if $T(n)$ is given by the recurrence*

$$T(n) = T(\alpha n) + T(\beta n) + \Theta(n),$$

then $T(n) = \Theta(n)$.

It is important here that the sum of α and β be *strictly* less than one. The lemma does not hold otherwise.

Proof Clearly, $T(n) = \Omega(n)$. We show that $T(n) = O(n)$ via the substitution method. For the induction, we assume that n is large enough, and that $T(n') \le cn'$ for all $n' < n$, where $c > 0$ is some constant. We prove that it follows that $T(n) \le cn$. We have, for some constant $a > 0$,

$$\begin{aligned} T(n) &\le T(\alpha n) + T(\beta n) + an \\ &\le c\alpha n + c\beta n + an \\ &= (c(\alpha+\beta) + a)n \\ &\le cn, \end{aligned}$$

provided $c(\alpha + \beta) + a \leq c$, or equivalently,

$$c \geq \frac{a}{1 - \alpha - \beta}.$$

Since the denominator is positive, the right-hand side is a finite number, so we can set $c = a/(1 - \alpha - \beta)$ to satisfy the inequality. □

The Algorithm

The linear-time Select algorithm is the same as RandomizedSelect, above, but where we replace RandomPartition(A, p, r) with the following procedure:

1. Let $n = r - p + 1$ be the number of items in the array. Divide $A[p \ldots r]$ into blocks $B_1, \ldots, B_k$ of five contiguous elements each. Thus $k = \lceil n/5 \rceil$, and $B_i = A[(p + 5i - 5) \ldots (p + 5i - 1)]$ for $1 \leq i < k$, and $B_k = A[(p + 5k - 5) \ldots r]$, which may have fewer than five elements if n is not a multiple of five.

2. For all $1 \leq i \leq k$, find the median m_i of block B_i. Put $m_1, \ldots, m_k$ into a separate array $B[1 \ldots k]$. (Note that this takes constant time per block, since each block has at most five elements).

3. Recursively call Select$(B, 1, k, \lceil k/2 \rceil)$ to find the median x of $m_1, \ldots, m_k$.

4. Partition $A[p \ldots r]$ as usual, using x as the pivot. Return the index q where x is placed after partitioning.

Let $T(n)$ be the running time of Select(A, p, r, i), where $n = r - p + 1$. The partitioning procedure above clearly takes linear time (in n) except for the recursive call to Select, which runs on an array of size about $n/5$.

How close is x to the median of $A[p \ldots r]$? Let's count the minimum number of elements in $A[p \ldots r]$ that are less than x. Since x is the median of $m_1, \ldots, m_k$, about $k/2$ of the m_i are less than x. But since each of these m_i is the median of a block of five elements there are at least two more elements of the array A less than m_i, so there are at least roughly $3k/2$ or about $3n/10$ elements of A less than x. It follows that there are *at most* $7n/10$ elements greater than x. A symmetrical argument shows that there are at least $3n/10$ elements of A greater than x, so there are at most $7n/10$ elements less than x. So in the worst case, when Select recurses on one of its sublists after partitioning, that sublist will have size at most $7n/10$. Thus in the worst case, the time for Select satisfies the following recurrence:

$$T(n) = T(n/5) + T(7n/10) + \Theta(n).$$

Letting $\alpha = 1/5$ and $\beta = 7/10$, we see that $\alpha + \beta = 9/10 < 1$, so we apply the lemma to get that $T(n) = \Theta(n)$. Thus Select takes linear time in the worst case.

There was a bit of slop in the previous analysis. We ignored round-off errors, for instance. Taking these into account, one can show that there are constants a, b such that the worst-case time satisfies

$$T(n) \leq T(n/5 + a) + T(7n/10 + b) + \Theta(n).$$

One can still apply the lemma in this case, however. By letting α' be just slightly bigger than α and β' just slightly bigger than β, we still have $\alpha' + \beta' < 1$, and

$$T(n) \leq T(\alpha' n) + T(\beta' n) + \Theta(n)$$

for large enough n, since the additive constants a and b are absorbed. In our particular case, we can set $\alpha' = \alpha + 1/30 = 7/30$ and $\beta' = \beta + 1/30 = 11/15$, and we still have $\alpha' + \beta' = 29/30 < 1$.

Lecture 13
Pointers and Records Using Arrays
Explicit Memory Management
Representing Trees
Binomial Coefficients

Explicit pointer and record types are not required to implement linked structures. Any programming environment that supports arrays (and there are some that only do this) will do. One can mechanically convert any program to an equivalent program without pointers or records. We'll show how this is done in two steps: first removing explicit pointer types, then removing record types.

Doing Without Pointers

Pointers are typically used to support linked structures, in which case the pointers point to records. To do without pointer types, one must do one's own memory management, rather than letting the system do it for you. There are some limitations to this (discussed below), but for most algorithms these limitations are not a problem. You declare an array of records big enough to hold the maximum number of records you expect to have active at any one time. This will serve as a surrogate to the systems dynamic memory store (often called the "heap," although it has no relation to priority queues). Then, instead of an explicit pointer to a record, you use the (integer) index of the record in the array. So pointer types become integer types in link fields or any other pointer variables, and dereferencing becomes array indexing.

We must do our own memory management in this case, simulating dynamic allocation (`malloc` in C, or `new` in C++) and deallocation (`free` in C, or `delete` in C++). Even if the system does support pointers and dynamic memory allocation, doing our own memory management is sometimes more efficient, both timewise and memorywise. Suppose we have the following declarations in C++:

```
struct node {
    double data;
    struct node * link;
}

struct node * my_list = 0;
```

Here, the pointer value 0 (the null pointer) is guaranteed not to point to any available memory, and thus `my_list` is initialized as an empty linked list. We replace the above with the following declarations (any identifier not explicitly declared or defined is assume to be previously):

```
struct node {
    double data;
    int link;
}
```

```
struct node heap[MAX_NUM_RECS];
int heap_size = 0;
int free_list = -1;

int my_list = -1;
```

Note that the pointer type has been changed to `int`. Analogous to the first declaration above, the value -1 (the "null" index) is guaranteed not to be a legal index into the array. The heap array will hold records that we dynamically allocate. The `heap_size` variable keeps track of how much of the heap has been used so far (initially none). The `free_list` variable will be explained below.

To allocate a new record, we could use `heap[heap_size]`, then increment `heap_size`. Unfortunately, this only allows a fixed finite number of allocations, and does not take advantage of reusing records that are previously deallocated. To fix this, we could tag records that we deallocate, then search the heap linearly for a tagged record when we want to allocate a new one. This allows recycling of old records, but the time-cost of linearly searching the heap each time we want a new record is prohibitively high.

The best approach is to maintain a (linked) list of deallocated nodes in the heap. Inserting and removing from this list (at the front) takes $O(1)$ time, so this is very efficient (the best we can do, asymptotically). The `free_list` variable points to this list. So to deallocate a record, we insert it into the free list. To allocate a record, we first check if one is available on the free list and use that. Only when the free list is empty do we resort to incrementing `heap_size`. Thus deallocation and allocation are implement as follows:

```
void deallocate(int p)
/* Deallocates a node pointed to by p */
{
    heap[p].link = free_list;
        /* instead of
           p->link = free_list */
    free_list = p;
}

int allocate()
/* Return a pointer to a newly allocated record,
   or a null pointer if no memory is available */
{
    int ret;
    if (free_list != -1) {
        ret = free_list;
        free_list = heap[free_list].link;
            /* instead of
               free_list = free_list->link */
        return ret;
    }
    if (heap_size < MAX_NUM_RECS)
        return heap_size++;
```

```
    return -1; /* out of memory */
}
```

This explicit memory management strategy can also be used with system dynamic allocation, where you maintain your own free list. In the current situation with arrays, it works great provided you know in advance roughly how many records you'll need at any given time. If you don't know this, however, or you have two or more different record types (maybe of different size) competing for memory, then this scheme is less useful than system allocation. The operating system uses a more sophisticated heap management system, whereby different sized chunks of memory can be allocated. These chunks are kept on a doubly linked circular list. If two deallocated chunks of memory are seen to be contiguous, then they are combined into one big chunk, which is more useful than two smaller chunks. Despite this, heaps can still be subject to fragmentation if two or more different sizes of records are used. System heap management is an interesting topic that every student should know about, since it can significantly influence the performance of algorithms that use dynamic memory. Sometimes it should be avoided because of fragmentation problems, where an explicit management scheme tailored to the problem at hand would work better.

Doing Without Records

Instead of an array of records, one can declare multiple arrays, one for each field of a record. So the heap and struct declarations above become simply

```
double data[MAX_NUM_RECS];
int link[MAX_NUM_RECS];
```

The data originally kept in the record at index i is now kept separately at `data[i]` and at `link[i]`. Deallocation and allocation thus become

```
void deallocate(int p)
/* Deallocates a node pointed to by p */
/* Similar to free() in C or delete in C++ */
{
    link[p] = free_list;
    free_list = p;
}

int allocate()
/* Return a pointer to a newly allocated record,
   or a null pointer if no memory is available */
/* Similar to malloc() in C or new in C++ */
{
    int ret;
    if (free_list != -1) {
        ret = free_list;
        free_list = link[free_list];
        return ret;
    }
    if (heap_size < MAX_NUM_RECS)
        return heap_size++;
```

```
    return -1; /* out of memory */
}
```

Doing Without Multidimensional Arrays

If we were only allowed linear arrays (of simple types) we could still simulate multidimensional arrays by storing the multidimensional array in a linear array in row-major order. Intead of

```
double matrix[HEIGHT][WIDTH];
int i, j;
/* ... */
matrix[i][j] = 3.14;
```

we would thus have

```
double matrix[HEIGHT*WIDTH];
int i, j;
/* ... */
matrix[WIDTH*i+j] = 3.14;
```

This is exactly how multidimensional arrays are stored and indexed in memory, anyway.

Trees

Hopefully, all these definitions are more or less familiar.

A *tree* is an undirected, connected, acyclic graph. If T is a tree, then the *size* of T is the number of vertices (nodes) in T. We will only consider trees of finite size. A tree of size $n > 0$ always has exactly $n - 1$ edges. This is worth stating as a theorem and proving. First, we define the *degree* of a vertex to be the number of vertices adjacent to it, i.e., the number of edges incident to it.

Theorem 11 *Any tree with $n > 0$ vertices has exactly $n - 1$ edges.*

Proof We go by induction on n. If $n = 1$, then the only tree on one vertex consists of that vertex and no edges. Thus the theorem is true for $n = 1$. Now assume that $n > 1$ and that the statement of the theorem holds for all trees of size $n - 1$. Let T be any tree of size n.

Claim 12 *T has a vertex of degree 1 (i.e., a leaf).*

Proof[of Claim] First, since T is connected with at least two vertices, it cannot have a vertex of degree 0. Suppose then that there is no vertex in T with degree 1. Then every vertex in T has degree at least 2. Start at any vertex v_1 and follow some edge to some other vertex v_2. Since v_2 has degree ≥ 2, there is some other edge to follow from v_2 to some new vertex v_3. We can continue in this way, building a path $v_1, v_2, v_3, \ldots$ by following new edges each time, but since T is finite, we must eventually come back to a vertex already on the path, i.e., $v_i = v_j$ for some $i < j$. Thus we have a cycle $v_i, v_{i+1}, v_{i+2}, \ldots, v_j = v_i$, which is impossible since T is a tree. Thus T must have a vertex of degree 1. □

Let v be a vertex of T with degree 1. If we remove v and its only edge from T, then the resulting graph T' is clearly still connected and acyclic, so it is a tree. Further, T' has size $n - 1$, so by the inductive hypothesis, T' has exactly $n - 2$ edges. We obtained T' from T by removing one vertex and one edge, so T must have exactly $n - 1$ edges. □

A *rooted* tree is a tree T that is either empty (no nodes) or has a distinguished node, called the *root* and denoted *root*[T] (if T is empty, then *root*[T] is a null reference). There is then a unique path from the root to any node. The *depth* of a node p in T is the length of the path from the root to p, and the *depth* of T is the maximum depth of any node in the tree (if the tree is empty, we'll take the depth to be -1). For nodes p and q in T, we say that p is an *ancestor* of q, or that q is a *descendant* of p, if p lies along the unique path from *root*[T] to q. If, in addition, $p \neq q$, then we say that p is a *proper* ancestor of q or that q is *proper* descendant of p. The *subtree of* T *rooted at* p is the tree of all descendants of p.

If T is a rooted tree and p is a nonroot node in T, then the *parent* of p is the (unique) immediate predecessor of p on the path from the root. The *children* of p are all nodes that share p as a parent. A *leaf* is a node with no children. The root has no parent. The parent of the parent is the *grandparent*; a child of a child is a *grandchild*, etc. The *height* of a node p is the length of the longest path from p to a leaf. The height of T is the height of its root, which is the same as the depth of T. (The height of an empty tree is -1 by convention, as with the depth.)

A rooted tree T is *ordered* if there is a linear order ("left to right") imposed on the children of each node. A *binary tree* is a rooted, ordered tree where each node has at most two children: the *left* child and the *right* child—roots of the *left* and *right* subtrees, respectively (if a child is missing, then that subtree is empty).

Representing Trees

Except for binary heaps, trees are usually represented as linked structures with information and links at each node. A node p of a binary tree, as well as containing a reference to satellite data, typically contains three links: to the parent (*parent*[p]), to the left subtree (*left*[p]), and to the right subtree (*right*[p]). *parent*[*root*[T]] is always a null pointer. In many applications, the *parent* field is not needed at any node, and can be omitted to save space.

For an arbitrary rooted ordered tree (not necessarily binary), we cannot use a fixed number of child fields, since we may not have any *a priori* bound on the number of children a node may have. Instead, each node has a pointer to a simple linked list of its children. Thus, together with a reference to satellite data, the three links of a node p are *parent*[p], *leftmost*[p] (pointing to the leftmost child, if there is one), and *rightsibling*[p] (pointing to p's immediate right sibling, if there is one).

Binomial Coefficients

For any integer $k \geq 0$ and any n (which could be any real number or even complex number), we define the *binomial coefficient*

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}.$$

We will only consider the case where n is an integer and $0 \leq k \leq n$. In this case,

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \binom{n}{n-k}.$$

$\binom{n}{k}$ is the number of ways to pick a subset of k elements from a set of n elements. It also figures prominently in the *Binomial Theorem*

Theorem 13 (Binomial Theorem) *For any real or complex x and y, and any integer $n \geq 0$,*

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}.$$

The Binomial Theorem immediately gives us some identities:

$$\sum_{k=0}^{n} \binom{n}{k} = (1+1)^n = 2^n,$$

and

$$\binom{n}{0} - \binom{n}{1} + \cdots \pm \binom{n}{n} = (1-1)^n = 0.$$

The first identity is also evident because both sides are the number of ways to choose a subset (of any size) of a set of n elements.

A Recurrence for Binomial Coefficients

For any integer $n \geq 0$, we have

$$\binom{n}{0} = \binom{n}{n} = 1,$$

and for integer k with $0 < k < n$, we have

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

These equations are immediate from the definition.

Some Inequalities

A sometimes useful inequality is

$$\binom{n}{k} = \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right) \geq \left(\frac{n}{k}\right)^k.$$

An upper bound on $\binom{n}{k}$ comes from Stirling's Approximation, which implies that $k! \geq (k/e)^k$. Whence,

$$\binom{n}{k} \leq \frac{n^k}{k!} \leq \left(\frac{en}{k}\right)^k.$$

Actually, it can be shown that

$$\binom{n}{k} \leq \frac{n^n}{k^k (n-k)^{n-k}},$$

and this is a good approximation. Letting $k = \lambda n$ for some $0 \leq \lambda \leq 1$, we get

$$\begin{aligned}
\binom{n}{\lambda n} &\doteq \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}} \\
&= \left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^n \\
&= 2^{nH(\lambda)},
\end{aligned}$$

where $H(\lambda) = -\lambda \lg \lambda - (1-\lambda)\lg(1-\lambda)$ is the *binary entropy* of λ. (Our convention is always that $0 \lg 0 = 0$.)

The Beads Problem

You have n boxes labeled $1, \ldots, n$ and k identical beads. How many distinct ways can you put the k beads into the n boxes, where each box can hold at most one bead? The answer is $\binom{n}{k}$. Now, suppose that a box can hold any number of beads. Now how many ways are there? Imagine the boxes lined up in a row, with $n-1$ dividers between them. Then any arrangement of beads in boxes is represented uniquely by a string such as

|*||*|***|||******|*

Here, the beads are represented as asterisks (*), and the dividers by vertigules (|). This string represents nine boxes and 16 beads, with two beads in the first box, three in the second, none in the third, one in the fourth, etc. So there is a one-to-one matching between arrangements of k beads in n boxes and strings consisting of k asterisks and $n-1$ vertigules. The number of such strings is evidently $\binom{n+k-1}{k}$.

Lecture 14
Probability Theory
Hash Tables

All the laws of probability can be derived from a few basic axioms. We start with a *sample space*, which is just an arbitrary set S. The elements of S are called *elementary events*. (We will assume here that S is *discrete*, i.e., S is either finite or countably infinite. One could be more general.) An *event* is any subset of S. As is customary, we write $\mathcal{P}(S)$ for the set of all events. The empty set $\emptyset$ is called the *null event*. Two events $A, B \subseteq S$ are *mutually exclusive* if $A \cap B = \emptyset$.

A *probability distribution* on S is a mapping $\Pr : \mathcal{P}(S) \to \mathbb{R}$, mapping events to real numbers, that satisfies the following axioms:

1. $\Pr[A] \geq 0$ for any event $A \subseteq S$.
2. $\Pr[S] = 1$.
3. $\Pr[A \cup B] = \Pr[A] + \Pr[B]$ for any two mutually exclusive events $A, B \subseteq S$. More generally, if $A_1, A_2, \ldots$ is some finite or countably infinite sequence of pairwise mutually exclusive events, then
$$\Pr\left[\bigcup_i A_i\right] = \sum_i \Pr[A_i].$$

We say that $\Pr[A]$ is the *probability* of the event A, or the probability that A occurs. For $x \in S$, we will abuse notation and write $\Pr[x]$ for $\Pr[\{x\}]$, the probability of the elementary event x. Axiom three implies that the probability of any event is the sum of the probabilities of its members (elementary events).

Here are some easy consequence of the axioms. For any event A, we let $\overline{A}$ denote the complement $S - A$ of A in S.

- $\Pr[\emptyset] = 0$. This is because $\emptyset \cap \emptyset = \emptyset$, that is, $\emptyset$ is mutually exclusive with itself. Applying the third axiom, we get
$$\Pr[\emptyset] = \Pr[\emptyset \cup \emptyset] = \Pr[\emptyset] + \Pr[\emptyset].$$
Subtract $\Pr[\emptyset]$ from both sides.
- $\Pr[\overline{A}] = 1 - \Pr[A]$ for any event A. Exercise.
- $\Pr[A \cup B] = \Pr[A] + \Pr[B] - \Pr[A \cap B]$ for any events A, B. Exercise.
- If $A \subseteq B$ then $\Pr[A] \leq \Pr[B]$. Exercise.
- $\Pr[A] \leq 1$ for any event A.

Conditional Probability

Let A and B be events with $\Pr[B] > 0$. Define

$$\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]}.$$

This is the *conditional probability of A given B*. (Look at a Venn diagram.) Note that $\Pr[B \mid B] = 1$. This is the probability we would get if we restrict our sample space from S to B, redefine events accordingly, and rescale the probability distribution to satisfy the second axiom.

Dependence

Events A and B are *independent* if $\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$. If $\Pr[B] > 0$, then clearly, A and B are independent iff $\Pr[A] = \Pr[A \mid B]$. Thus, conditioning on B does not affect the probability of an independent event A.

$A_1, A_2, \ldots$ are *mutually independent* if $\Pr[A_1 \cap A_2 \cap \cdots] = \Pr[A_1] \cdot \Pr[A_2] \cdot \cdots$.

Examples

Let S represent two independent flips of a fair coin, i.e., $S = \{H, T\} \times \{H, T\}$ (H for heads, T for tails), and for any $a, b \in \{H, T\}$ we set $\Pr[(a, b)] = 1/4$. Let A be the event, "the first flip is heads." Thus $A = \{(H, H), (H, T)\}$. Let B be the event, "the two flips are different." Thus $B = \{(H, T), (T, H)\}$. We have $\Pr[A] = \Pr[B] = 1/2$, and

$$\Pr[A \cap B] = \Pr[(H, T)] = 1/4 = \Pr[A] \cdot \Pr[B],$$

so A and B are independent.

The following easy theorem relates $\Pr[A \mid B]$ with $\Pr[B \mid A]$. It forms the basis of Bayesian analysis.

Theorem 14 (Bayes's Theorem) *If A and B are events with positive probability, then*

$$\Pr[A \mid B] = \frac{\Pr[A] \Pr[B \mid A]}{\Pr[B]}.$$

A *p-biased coin flip* is given by the sample space $S = \{H, T\}$, with $\Pr[H] = p$ and $\Pr[T] = q = 1 - p$, where p is some real number with $0 \leq p \leq 1$. If we flip a p-biased coin n times (independently), then the sample space is $\{H, T\}^n$, and

$$\Pr[(a_1, a_2, \ldots, a_n)] = \prod_{i=1}^{n} \Pr[a_i],$$

where each $a_i \in \{H, T\}$ and each term of the product is either p or $1-p$. For $0 \leq k \leq n$, there are $\binom{n}{k}$ many tuples with exactly k heads, and the probability of each such tuple is $p^k(1-p)^{n-k} = p^k q^{n-k}$. Thus,

$$\Pr[\text{exactly } k \text{ heads}] = \binom{n}{k} p^k q^{n-k},$$

which is just the kth term in the binomial expansion of $1 = 1^n = (p+q)^n$. If $p = 1/2$, then we call the coin *unbiased* or *fair*, and we have $\Pr[k \text{ heads}] = 2^{-n}\binom{n}{k}$.

Random Variables

As above, we assume a sample space S with a probability distribution $\Pr : S \to \mathbb{R}$ satisfying the usual axioms.

A *random variable* over S is any mapping $X : S \to \mathbb{R}$. (Thus the probability distribution is itself a random variable.) The *expectation value* of a random variable X is defined to be

$$E(X) = \sum_{a \in S} X(a) \Pr[a].$$

Linearity of Expectation

For any two random variables X and Y and any real constant c, we have

$$E(cX + Y) = cE(x) + E(Y).$$

For example, suppose $0 \leq p < 1$ and we flip a p-biased coin until we see tails. Then the sample space is

$$S = \{T, HT, HHT, HHHT, \ldots, H^kT, \ldots\},$$

with $\Pr[H^kT] = p^k q$, where $q = 1 - p$. Note that this really is a probability distribution, since

$$\sum_{k=0}^{\infty} p^k q = q \sum_{k=0}^{\infty} p^k = \frac{q}{1-p} = 1.$$

Let random variable X be the number of heads before the first tail. That is, $X(H^kT) = k$ for all $k \geq 0$. What is $E(X)$?

$$\begin{aligned} E(X) &= \sum_{k=0}^{\infty} X(H^kT) \Pr[H^kT] \\ &= \sum_{k=0}^{\infty} k p^k q \\ &= q \sum_{k=0}^{\infty} k p^k \\ &= \frac{qp}{(1-p)^2} \\ &= \frac{p}{q}. \end{aligned}$$

Indicator Random Variables

If A is an event, then there is a natural random variable associated with A, namely $I[A]$, the *indicator random variable of* A. The function $I[A]$ maps an elementary event x to 1 if $x \in A$, and to 0 otherwise. Notice that $E(I[A])$ is just $\Pr[A]$.

Hash Tables

A *hash table* is an array $T[1 \ldots m]$ (in this case we say that T has m slots) together with a function h mapping key values to integers in the range $1 \ldots m$. The function h, called the *hash function*, is assumed to be easy to compute. We would also like h to look as "random" as possible. We store n elements into the hash table T. To insert an item with key k, we first compute $i = h(k)$, then place the item at $T[i]$. We may have *collisions*, i.e., two or more different items mapping (or "hashing") to the same index. Collisions can be resolved by either *chaining* or *open addressing*. We'll only discuss chaining, which is maintaining at each location $T[i]$ a linked list of all the items inserted into T whose keys hash to i.

We'd like to analyze the performance of a hash table, assuming that h maps keys to indices uniformly at random, independently for different keys. For a hash table with m slots containing n items, we define the *load factor*

$$\alpha = \frac{n}{m}.$$

This is the expected length of a linked list in the table. Note that with chaining it is possible that $n > m$, and so $\alpha > 1$.

We assume that h takes $\Theta(1)$ time to compute. Clearly then, the expected time for an unsuccessful search is $\Theta(1 + \alpha)$. The 1 is the time to compute h of the key, and the α is the time to traverse the corresponding linked list.

Analyzing a successful search is a bit harder, assuming any existing element is equally likely to be searched. This is because if we restrict our sample space to just the items that are already in the table, then any given item is more likely to be in a longer list than a shorter one. Nevertheless, the expected successful search time can still be shown to be $\Theta(1 + \alpha)$.

Assume that distinct keys $k_1, \ldots, k_n$ are inserted in order into an initially empty hash table with m slots, and let $\alpha = n/m$ as before. We'll assume that each key k is are inserted onto the front of the linked list of keys hashing to $h(k)$. This means that the time to successfully search for k is one plus the number of items inserted *after* k that hash to the same list as k. For $1 \leq i, j \leq n$, let X_{ij} be the event that $h(k_i) = h(k_j)$. Then we have

$$\begin{aligned}
& E(\text{successful seach time}) \\
&= E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} I[X_{ij}]\right)\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j} E(I[X_{ij}])\right) \\
&= \frac{1}{n}\sum_{i}\left(1 + \sum_{j}\frac{1}{m}\right) \\
&= 1 + \frac{1}{nm}\sum_{i}(n-i) \\
&= 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}.
\end{aligned}$$

The second equality follows by linearity of expectation.

Lecture 15
Binary Search Trees
Red-Black Trees

We all know what a Binary Search Tree (BST) is. Access, insertion, and deletion all take $\Theta(h)$ time in the worst case, where h is the height of the tree. At best, a binary tree of size $n > 0$ will have height $\lfloor \lg n \rfloor$ (e.g., the almost full tree used for the binary heap). At worst, the height is $n - 1$, and tree operations are no faster than with a linked list, i.e., linear time. This latter case occurs, for example, when items are inserted into the tree in order (increasing or decreasing). Note that Quicksort displays worst-case behavior with a sorted array (when the pivot is always chosen to be the first element of the list). This is not a coincidence; there is a close connection between the sequence of pivot choices and a preorder traversal of the corresponding BST built by inserting items from the array by increasing index.

By the way, the book allows duplicate keys in a BST. I don't allow them. If you really want duplicate keys (with different satellite data, of course), then store all duplicate keys at the same node, i.e., have the node point to a linked list of all the items with that key.

Although for random insertions and deletions, the expected height of a BST is $O(\lg n)$, this is not satisfactory, because inputs often don't look "random," i.e., some input orders (e.g., sorted or almost sorted) occur with above average frequency. Thus we would like a way to keep the *worst-case* time for tree operations to $O(\lg n)$, that is, keep the height of the tree $O(\lg n)$. This is typically done by performing structural alterations to the tree to balance it when it gets too out of balance. We need to keep some extra information at each node to help us detect when the tree is becoming too unbalanced.

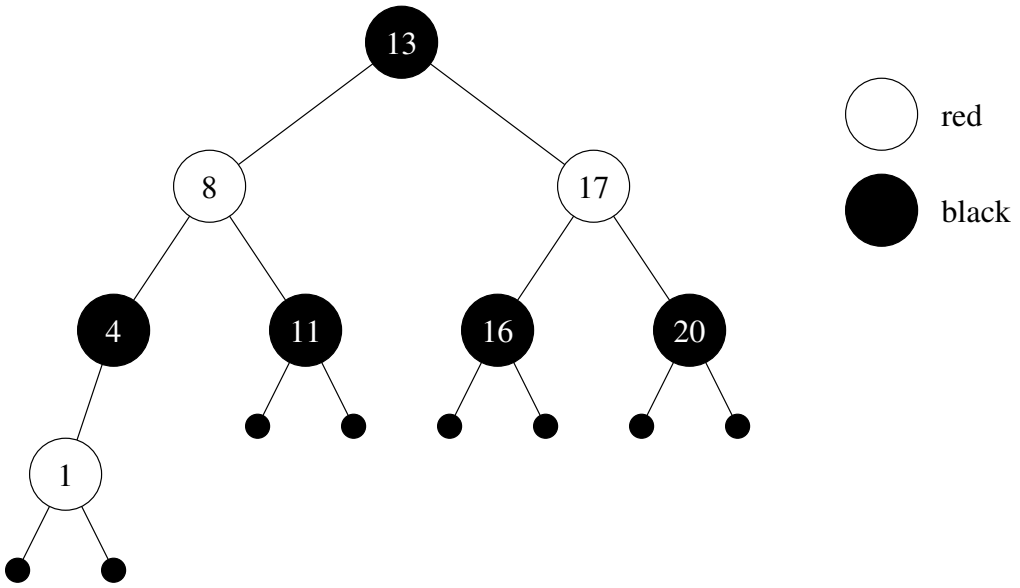
One such scheme is the AVL tree, which I cover in CSCE 350. Here we'll see a different scheme: red-black trees.

A red-black (RB) tree is a BST where each node at any given time has a color: red or black. If T is a RB tree, we assume that any null child pointer points to a phantom leaf node. Thus every key-bearing node is considered an internal node. A leaf has no information other than its color, which is always black. Thus we can implement leaves by a single shared black node, $nil[T]$, that is pointed to by any child pointer that would otherwise be null in an ordinary BST.

The conditions for a nonempty RB tree are as follows:

1. The root is always black.
2. Every leaf is black.
3. If a node is red, then both its children are black.
4. For any node p in the tree, all paths from p to leaves go through the same number of black nodes.

Here is a RB tree:



The small black nodes (dummy nodes) are the leaves (which can be implemented by a single shared node). The data are contained in the internal nodes. It is not necessary that all data-bearing nodes on the same level are the same color.

First we prove that a BST satisfying these conditions (the RB conditions) must have height $O(\lg n)$. We first define the *black height* $\mathrm{bh}(x)$ of a node x in a RB tree to be the number of black nodes encountered on some (any) path from x to a leaf, including the leaf, but not including x itself.

Lemma 15 *Let T be a red-black tree with n nodes with height h. Then*

$$h \leq 2\lg(n+1).$$

Proof We prove by induction on the height of a node x that the subtree rooted at x has size at least $2^{\mathrm{bh}(x)} - 1$. If x has height zero, then it is a leaf with black height 0 and subtree of size $1 \geq 0 = 2^0 - 1$, which satisfies the base case. Now assume that x has positive height, and let y and z be the children of x. Clearly, y and z have black height at least $\mathrm{bh}(x) - 1$, and since y and z have height less than the height of x, we can apply the inductive hypothesis to y and z. Let s_x, s_y, and s_z be the sizes of the subtrees rooted at x, y, and z, respectively. Then,

$$\begin{aligned} s_x &= s_y + s_z + 1 \\ &\geq (2^{\mathrm{bh}(y)} - 1) + (2^{\mathrm{bh}(z)} - 1) + 1 \\ &\geq (2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 1 \\ &= 2^{\mathrm{bh}(x)} - 1, \end{aligned}$$

which proves the induction.

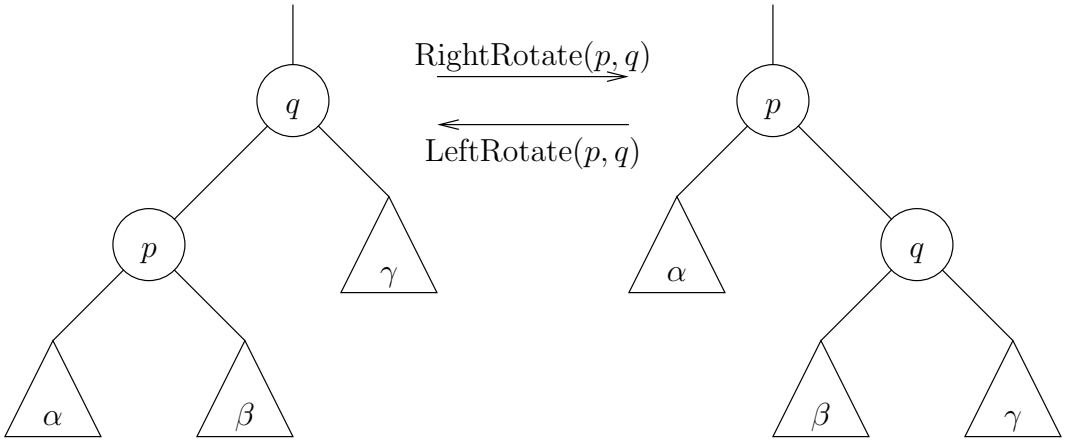
Now let r be the root of T. r has height h, and since no two adjacent nodes on any path from r to a leaf can be red, at least half the nodes on such a path must be black. Thus, $\mathrm{bh}(r) \geq h/2$. By the inequality we just proved, we have

$$n \geq 2^{\mathrm{bh}(r)} - 1 \geq 2^{h/2} - 1,$$

since n is the size of the subtree rooted at r, i.e., T itself. Solving this inequality for h proves the lemma. □

An empty tree contains just the single leaf node $nil[T]$. Insertion into a RB tree T happens in two phases. In the first we insert a node just as in the case for a BST. If T was empty before the insertion, then this first node inserted becomes the root, so it must be colored black. Otherwise, each inserted node is initially colored red. The second phase restores the RB conditions, which may have been violated after the first phase. This latter phase may only adjust colors of nodes, or it may structurally alter the tree. We look at this phase in more detail now.

The tree will be altered by means of *rotations*. There are left- and right-rotations. In a left-rotation, a parent p and its right child q move so that q takes the place of the parent and p becomes the left child of q. The three subtrees below p and q are reattached the only way they can to preserve the BST order of the tree. We'll call this operation LeftRotate(p, q). A right-rotation is just the inverse of a left-rotation. We'll call this RightRotate(p, q).



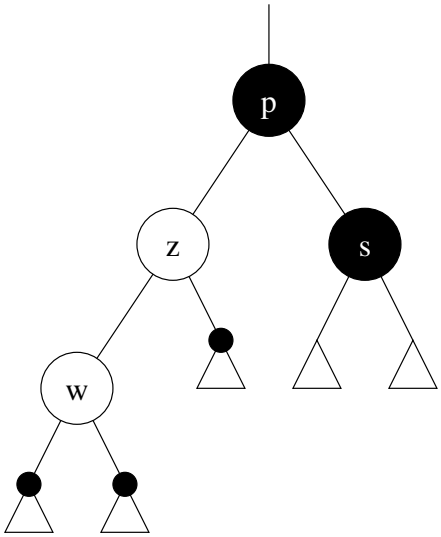
Note that the arguments to LeftRotate and RightRotate always come in increasing order.

Now for the second phase ("clean-up" or "fix-up"). Assume a node x is inserted into a nonempty RB tree T. Then x is not the root, so it has a parent $y = parent[x]$ and an initial color of red. Note that this does not affect the fourth RB condition. If y's color is black, there is nothing to do (both children of x are leaves and hence black). The problem is when y is red; then we have two red nodes in a row. We now assume this case.

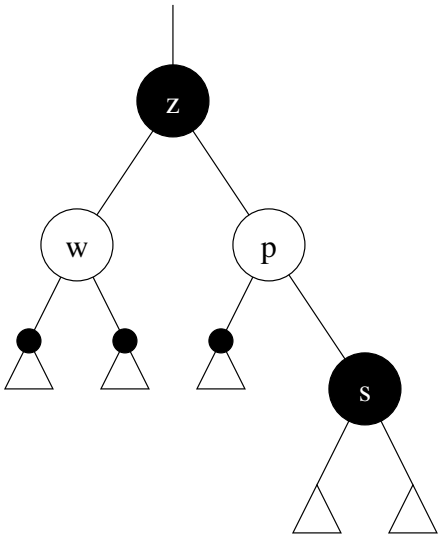
The node y is red, so it cannot be the root. We consider y's sibling s (x's uncle or aunt), and their common parent p (x's grandparent), which must be black. If s is red, then we: (i) switch the colors of both y and s to black, and (ii) switch the color of p to red (unless p is the root, in which case we're done). Note that we did not structurally alter the tree in this case, but merely changed colors of some nodes. Although we fixed the problem with x and y both being red, we may have created the same problem with p and its parent, which may also be red (p was black originally; now it's red). If this is the case then we do the entire operation again (calling FixUp recursively, say) with p and its parent in place of x and y, respectively. We make progress this way, because we get closer to the root.

Now suppose s is black. Here we will have to structurally alter the tree. We'll assume that y is the left child and s is the right child of p. If not, then we simply swap the roles of left and right in everything that follows.

If x is a right child of y, then we do a LeftRotate(y, x) so that now y is a left child of x; otherwise, we don't rotate. In either event, we now have a node z and its left child w both red (here, $\{z, w\} = \{x, y\}$). s (black) is the right sibling of z (red) with common parent p (black).



We do a RightRotate(z, p), so that p is now the right child of z. Finally, we make z black and p red.



A simple case analysis shows that the RB properties are restored by these operations, so there is nothing more to do.

Fixing up after regular BST insertion takes time $O(h)$ (h is the height of the tree) in the worst case (i.e., when we must propagate red parent-child pairs all the way up to the root), so it does not affect the total asymptotic running time for insertion. Note also that at most two rotations occur for any insertion.

Deletion from red-black trees is similar (first a regular BST deletion, then a fix-up), but there are more cases to consider, and we won't do it here.

Lecture 16
The AVL Condition
Augmented Structures

A binary tree satisfies the *AVL condition* at a node x if the heights of the right and left subtrees of x differ by at most one. The tree itself is an *AVL tree* if it satisfies the AVL condition at each of its nodes. Like red-black trees, AVL trees of size n are guaranteed to have height $\Theta(\lg n)$. So if we have a BST that is an AVL tree, searching is asymptotically optimal (i.e., $\Theta(\lg n)$) in the worst case. One can show (exercise) that one can maintain the AVL condition given insertions and deletions, and that the cost of maintenance of the condition does not asymptotically increase the time of insertion/deletion. Thus, AVL trees (like RB trees) are a worst-case asymptotically optimal way to implement BSTs.

Here we show that any AVL tree of size n has height $O(\lg n)$. We do this by considering the smallest possible size m_h of an AVL tree with height $h \geq -1$. Clearly, we have

$$\begin{aligned} m_{-1} &= 0, \\ m_0 &= 1. \end{aligned}$$

Now suppose $h > 0$, and consider an AVL tree T of height h and of minimum size m_h. T consists of a root r with left and right subtrees T_1 and T_2, respectively. At least one of T_1 and T_2 has height $h-1$; say T_2. Then T_2 has size m_{h-1}, since otherwise we can reduce the size of T (without changing its height) by reducing the size of T_2 (without changing its height). The possible heights of T_1 are then $h-1$ or $h-2$. Since T is as small as possible, T_1 must have the smaller height, i.e., $h-2$, and be of size m_{h-2} (if T_1 had height $h-1$, then deleting all the nodes on its bottom level would make T_1 a strictly smaller AVL tree of height $h-2$, thus reducing the size of T). We thus get the recurrence,

$$m_h = m_{h-2} + m_{h-1} + 1,$$

for all $h \geq 1$. This bears some resemblance to the Fibonacci sequence:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-2} + F_{n-1}, \end{aligned}$$

for $n \geq 2$. In fact,

Claim 16 *For all $h \geq -1$,*

$$m_h = F_{h+3} - 1.$$

Proof It suffices to show that the function $f(h) = F_{h+3} - 1$ satisfies the same relations as m_h, since those relations uniquely determine m_h. We have

$$f(-1) = F_2 - 1 = 1 - 1 = 0,$$

$$f(0) = F_3 - 1 = 2 - 1 = 1,$$

and for $h \geq 1$,

$$\begin{aligned} f(h) &= F_{h+3} - 1 \\ &= F_{h+1} + F_{h+2} - 1 \\ &= (F_{h+1} - 1) + (F_{h+2} - 1) + 1 \\ &= f(h-2) + f(h-1) + 1. \end{aligned}$$

Thus $f(h)$ satisfies the same relations as m_h, so $f(h) = m_h$ for all $h \geq -1$. □

In a previous exercise, you showed that $F_h = \Theta(\varphi^h)$, where $\varphi = (1+\sqrt{5})/2 \doteq 1.6\ldots$ is the Golden Ratio. Thus $m_h = \Theta(\varphi^{h+3}) = \Theta(\varphi^3\varphi^h) = \Theta(\varphi^h)$. Thus for any AVL tree of size n and height h, we have

$$n \geq m_h = \Theta(\varphi^h),$$

and hence

$$n = \Omega(\varphi^h).$$

Taking the $\log_\varphi$ of both sides gives

$$h = O(\log_\varphi n) = O(\lg n),$$

which is what we wanted to show.

Augmented Structures

Sometimes it is useful to add more information to data structure to allow it to do more things.

For example, consider the problem of dynamic order statistics. You have a collection of n keys that is changing over time, with keys being inserted and deleted. You wish to allow the usual dictionary operations, such as insert, delete, look-up, print in order, etc., so you use a BST, say, a red-black tree. Suppose you also wish to support the Select operation, which finds the kth smallest key, for input k. You can always do this in time $\Theta(k + \lg n)$, say, by cycling through the first k elements in order. This is fine if we expect that Select operations will only be done rarely. But if they are done frequently, we can do much better.

Maintain an additional field in each node that contains the size of the subtree rooted at that node. If we have this, then Select can easily be done in $O(\lg n)$ time. It is easy to show how this extra field can be updated correctly with insertion/deletion without affecting the asymptotic run times of those operations, even if rotations are allowed (e.g., for an AVL or red-black tree).

Lecture 17
Dynamic Programming

Sometimes a recursive implementation is too inefficient, because of redundant recursive calls. If the range of possible values passed to recursive calls is not too large, then it is usually better to build the solution bottom-up rather than solve it recursively top-down. This bottom-up technique, which usually involves filling in entries of a table, is called *dynamic programming*.

Example: computing Fibonacci numbers, binomial coefficients

The sequence of Fibonacci numbers $\{F_n\}_{n\geq 0}$ is given by the relations

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_n &= F_{n-2} + F_{n-1}, \end{aligned}$$

for $n \geq 2$. One could translate this directly into C++ code:

```
unsigned int fib(unsigned int n)
{
    if (n==0) return 0;
    if (n==1) return 1;
    return fib(n-2) + fib(n-1);
}
```

This code will correctly compute F_n, but it is very inefficient. For example, `fig(5)` calls `fib(3)` recursively twice, and so on. The number of redundant recursive calls is exponential in n. A much saner approach is to use the relations to fill up a table with the Fibonacci sequence:

```
unsigned int fib(unsigned int n)
{
    unsigned int *tab = new unsigned int[n+2];
    tab[0] = 0;
    tab[1] = 1;
    for (i==2; i<=n; i++)
        tab[i] = tab[i-2] + tab[i-1];
    return tab[n];
}
```

Ignoring overflow, this routine runs linearly in n. This is a classic example of dynamic programming. The code above can be improved further with the observation that only two adjacent Fibonacci numbers need to be remembered at any given time, thus we only need constant extra space:

```
unsigned int fib(unsigned int n)
{
    unsigned int current=0, next=1, temp;
    for (; n>0; n--)
    {
        temp = current + next;
        current = next;
        next = temp;
    }
    return current;
}
```

(Exercise: can you get by with only two local variables?)

Example: the Knapsack Problem

You have a knapsack with capacity c and you are given a collection of k drink containers of varying sizes $a_1, a_2, \ldots, a_k$. You want to determine if you can select some subset of the available containers to fill up your knapsack completely without overfilling it. You can assume that the containers are flexible so that they can assume any shape, thus only their sizes are relevant. You can also assume that c and all the a_i are positive integers. Thus the problem is equivalent to: given a list $a_1, \ldots, a_k$ of positive integers and a positive integer c, is there a sublist that adds up to c exactly? Equivalently, to there exist $b_1, \ldots, b_k \in \{0, 1\}$ such that

$$\sum_{i=1}^{k} b_i a_i = c?$$

This decision problem is known as the $0 - -1$ Knapsack problem, or the Subset Sum problem.

Here's a straightforward solution that involves recursive backtracking. The function Fill takes c and $\langle a_1, \ldots, a_k \rangle$ and returns a Boolean value.

```
Fill(c, ⟨a_1, ..., a_k⟩)
// Preconditions: c ≥ 0 and a_1, ..., a_k > 0 (all integers)
// Returns TRUE iff some sublist of the a_i adds exactly to c
    IF c = 0 THEN return TRUE
    IF k = 0 THEN return FALSE
    return Fill(c, ⟨a_1, ..., a_{k-1}⟩) OR
        (c − a_k ≥ 0 AND Fill(c − a_k, ⟨a_1, ..., a_{k-1}⟩))
```

Although correct, this solution suffers from the same redundancies that afflicted the recursive fib program above: there may be several calls to Fill$(d, \langle a_1, \ldots, a_i \rangle)$ for $d < c$ and $i < k$.

Here's a dynamic programming solution that builds a table of intermediate results, ensuring that each intermediate result is computed only once. We maintain an array $B[0 \ldots c, 0 \ldots k]$ of Booleans such that each $B[d, i]$ holds the value Fill$(d, \langle a_1, \ldots, a_i \rangle)$. We fill in the entries of B in order of increasing d and i.

```
Fill(c, ⟨a_1, ..., a_k⟩)
// Preconditions: c ≥ 0 and a_1, ..., a_k > 0 (all integers)
// Returns TRUE iff some sublist of the a_i adds exactly to c
    let B[0...c, 0...k] be an array of Boolean values
    B[0, 0]← TRUE
    FOR d := 1 TO c DO
        B[d, 0]← FALSE
    FOR i := 1 TO k DO
        FOR d := 0 TO c DO
            B[d, i]←B[d, i − 1]
            IF d − a_i ≥ 0 AND B[d − a_i, i − 1] THEN
                B[d, i]← TRUE
    return B[c, k]
```

This version follows the same logical rules as the recursive solution, but takes time and space $\Theta(ck)$, since that's the size of the table B, and each entry in B takes $O(1)$ time to compute. The time may be much less than the time taken by the recursive solution. The table is computed column

by column, and each successive column only requires the previous column. Thus the space can be reduced to that required for two adjacent columns: $\Theta(c)$. Can you get by with only one column? Yes, provided you fill in the entries of the column in *reverse*, i.e., for d running from c down to 0.

Example: optimal order for matrix multiplication

A $m \times n$ matrix is a two-dimensional array of numbers, with m rows (indexed $1 \ldots m$) and n columns (indexed $1 \ldots n$), for $m, n \geq 1$. Two matrices A and B can be multiplied together, forming the product AB, provided they are *compatible* which means that the number of columns of A is the same as the number of rows of B. Multiplying an $m \times n$ matrix A by an $n \times p$ matrix B yields an $m \times p$ matrix $C = AB$, where

$$C[i, j] = \sum_{k=1}^{n} A[i, k]B[k, j].$$

Thus the straightforward way of multiplying A with B requires mnp many scalar multiplications, which dominates the total running time.

Matrix multiplication is associative, i.e., $(AB)C = A(BC)$. (It is not commutative, however; it may be that $AB \neq BA$.) Although $(AB)C = A(BC)$ so the order of multiplication does not affect the result, the time taken for one order may be vastly different from the time taken by the other. For example, suppose A is 5×20, B is 20×2, and C is 2×10. Then computing $(AB)C$ requires

- $5 \cdot 20 \cdot 2 = 200$ scalar multiplications for computing AB, plus
- $5 \cdot 2 \cdot 10 = 100$ scalar multiplcations for matrix-multiplying the result by C,

for a total of 300 scalar multiplications. On the other hand, computing $A(BC)$ requires

- $20 \cdot 2 \cdot 10 = 400$ scalar multiplications for computing BC, plus
- $5 \cdot 20 \cdot 10 = 1000$ scalar multiplcations for matrix-multiplying the result by C,

for a total of 1400 scalar multiplications. Thus if we are multiplying lots of matrices together, we can get significant savings by choosing the order of multiplication wisely.

Suppose we have a list of matrices $A_1, \ldots, A_n$, where A_i and A_{i+1} are compatible, for $1 \leq i < n$. Thus we have integers $p_0, p_1, \ldots, p_n \geq 1$ such that A_i is $p_{i-1} \times p_i$. Given these values p_i, we would like to find a way to fully parenthesize $A_1 \cdots A_n$ to minimize the total number of scalar multiplcations. This is a job for dynamic programming!

For $1 \leq i \leq j \leq n$, we will let $A_{i \cdots j}$ denote the product $A_i A_{i+1} \cdots A_j$. Note that $A_{i \cdots j}$ has dimensions $p_{i-1} \times p_j$. Suppose $i < j$ and that $A_i \cdots A_j$ has been optimally parenthesized. Then the last matrix multiplication to be performed will be multiplying some $A_{i \cdots k}$ by $A_{(k+1) \cdots j}$, where $i \leq k < j$ is chosen to minimize the overall number of scalar multiplications. This last matrix multiplication alone requires $p_{i-1} p_k p_j$ scalar multiplications. Further, $A_{i \cdots k}$ and $A_{(k+1) \cdots j}$ must both be parenthesized optimally, for otherwise an improved parenthesization of one or the other will lead to a better parenthesization of $A_{i \cdots j}$ overall. Let $m(i, j)$ be the optimal number of scalar multiplications needed to compute $A_{i \cdots j}$. From the considerations above, we have

$$m(i, j) = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \leq k < j} [m(i, k) + m(k + 1, j) + p_{i-1} p_k p_j) & \text{if } i < j. \end{cases}$$

The optimal number of scalar multiplications for computing $A_{1\cdots n}$ is then $m(1,n]$). Note that each $m(i,i) = 0$, because $A_{i\cdots i}$ is just A_i, and so there is nothing to compute.

A naive implementation of $m(i,j)$ as a recursive function would take exponential time. Instead, we note that there are only $n(n+1)/2$ many different m-values that need to be computed. Thus we can compute the m-values by putting them into a reasonably sized array $m[1\ldots n, 1\ldots n]$. Each diagonal element will be 0. Whenever we compute $m[i,j]$ with $i < j$, we already have computed $m[i,k]$ and $m[k+1,j]$ for all $i \le k < j$, so computing $m[i,j]$ is straightforward. In this case, we also wish to keep track of the value of k that achieves the minimum, since this will tell us where $A_{i\cdots j}$ can be optimally split. We could put these k values in another two-dimensional array, or we could just use $m[j,i]$ to hold these values, since these entries are not being used for m-values.

```
MatrixChainOrder(p, n)
// p[0...n] is an array of positive integers.
// Returns an m-array as described above.
    let m[1...n, 1...n] be an integer array
    for i←n downto 1 do
        m[i,i] = 0
        for j←i + 1 to n do // compute m[i,j]
            m[i,j]←∞
            for k←i to j − 1 do
                s←m[i,k] + m[k+1,j] + p[i−1]p[k]p[j]
                if s < m[i,j] then
                    m[i,j]←s
                    m[j,i]←k
    return m
```

What is the running time of this procedure? How much space is used? Assume that the matrix entries do not get very large.

Example: longest common subsequence

Two finite sequences of objects $X = \langle x_1, \ldots, x_m\rangle$ and $Y = \langle y_1, \ldots, y_m\rangle$ can be considered similar if they have a long subsequence in common. For example, if two strands of DNA have a long sequence of base pairs in common, then they are likely to have descended from a recent common ancestor. We'd like to find the length of a longest common subsequence (LCS) of X and Y.

A *subsequence* of a sequence X is any sequence obtained from X by removing zero or more elements from X. Thus $\langle z_1, \ldots, z_k\rangle$ is a subsequence of $\langle x_1, \ldots, x_m\rangle$ iff there exist indices $1 \le i_1 < i_2 < \cdots < i_k \le m$ such that $z_j = x_{i_j}$ for all $1 \le j \le k$.

Here is a recursive solution to the problem of finding an LCS of X and Y. If $m = 0$ or $n = 0$, that is, one of the sequence is empty, then clearly the only LCS is the empty sequence, of length 0. This is the base case. Now suppose $m, n > 0$, and consider the prefixes X_{m-1} and Y_{n-1} of X and Y, respectively. (If Z is any sequence, we let Z_i denote the sequence containing the first i elements of Z.) If $x_m = y_n$, then clearly, any LCS of X and Y must end in this common value, and be preceded by an LCS of X_{m-1} and Y_{n-1}. If $x_m \ne y_n$, then any LCS of X and Y includes at most one of x_m and y_n as its last element. Thus, such an LCS must be either an LCS of X and Y_{n-1} or an LCS of X_{m-1} and Y, whichever is longer.

Let $c[i,j]$ be the length of an LCS of X_i and Y_j, for $0 \le i \le m$ and $0 \le j \le n$. Then,

$$c[i,j] = \begin{cases} 0 & \text{if } ij = 0, \text{ else} \\ c[i-1,j-1]+1 & \text{if } x_i = y_j, \\ \max\{c[i,j-1], c[i-1,j]\} & \text{otherwise.} \end{cases}$$

Lecture 18
LCS (continued)
Greedy Algorithms: Huffman Codes

The recurrence relations for $c[i,j]$ make them easy to compute with dynamic programming, using nested for-loops for i and j where each index is increasing. This algorithm takes $\Theta(mn)$ time and space, but the space can be reduced to $\Theta(\min(m,n))$.

How can we recover an actual LCS if we want one? We keep track of how each $c[i,j]$ is obtained. If $x_i = y_j$, then we label $c[i,j]$ with a diagonal arrow ($\nearrow$); otherwise, we label $c[i,j]$ with either a left arrow ($\leftarrow$) or an up arrow ($\uparrow$), whichever points to the entry equal to $c[i,j]$ (if there is a tie, either arrow can be used; this reflects the fact that there may be more than one LCS, or that the same LCS can be obtained in more than one way). When the table calculation is complete, we build an LCS by following the arrows back from $c[m,n]$, prepending x_i onto the LCS we are building whenever a diagonal arrow is encountered at some row i. One proves that this is indeed an LCS by induction on m, n, much the same way as the proof that the relations obeyed by $c[i,j]$ imply correctness of the $c[i,j]$.

Greedy Algorithms

For some optimization problems, even dynamic programming can be overkill. When reducing a problem for a recursive solution, one typically has a choice of possible reductions, one of which eventually leads to a globally optimal solution. Sometimes we don't know which choice is the right one, so we must try all of them (by dynamic programming, say). Sometimes, however, if one chooses a *locally* optimal reduction (by some measure) each time, then this provably leads to a global optimum. A greedy algorithm is one that follows this strategy, i.e., it always picks a locally optimal step, or more informally, a step that looks best at the time. If correct, this can lead to significant speed-up over trying all possible reductions.

Huffman Codes

We'll only look at one example of a greedy algorithm: constructing an optimal binary prefix code for an alphabet given that each letter has a frequency. Such a code is called a Huffman Code. Huffman codes are used for data compression. Given an alphabet $C = \{c_1, \ldots, c_n\}$, a *binary prefix code for* C is a mapping $\varphi : C \to \{0,1\}^*$ such that $\varphi(c_i)$ is never a prefix of $\varphi(c_j)$ for any $i \ne j$. Elements of the range of φ are called *codewords*. We can extend φ to encode a string $\sigma = c_{i_1} c_{i_2} \cdots c_{i_k} \in C^*$ as the concatenation

$$\varphi(\sigma) = \varphi(c_{i_1})\varphi(c_{i_2}) \cdots \varphi(c_{i_k}) \in \{0,1\}^*.$$

Since no codeword is a prefix of any other codeword, the string σ can be uniquely recovered from $\varphi(\sigma)$ (and a description of φ itself). Such a property of a code is called *unique decodability*. All

prefix codes are thus uniquely decodable, further, the decoding can be done in real time by reading the code once left to right.

Example

Suppose $C = \{a, b, c, d\}$. A simple prefix code for C is to encode each character as two bits:

$$\begin{aligned} a &\mapsto 00 \\ b &\mapsto 01 \\ c &\mapsto 10 \\ d &\mapsto 11 \end{aligned}$$

Suppose $\sigma \in C^*$ is a string of length 1000. Then this code encodes σ by a string of 2000 bits. If the frequencies of the letters in σ are not uniform, however, we may be able to find a better code (one whose encoding of σ is shorter). Suppose a occurs in σ about half the time, c occurs about one quarter of the time, and b and d each occur about one eighth of the time. Thus we have the following letter frequences for σ:

letter	**frequency**
a	500
b	125
c	250
d	125

Suppose we use the following prefix code instead:

$$\begin{aligned} a &\mapsto 0 \\ b &\mapsto 110 \\ c &\mapsto 10 \\ d &\mapsto 111 \end{aligned}$$

Then σ is now encoded by a bit string of length $500 \cdot 1 + 250 \cdot 2 + 125 \cdot 3 + 125 \cdot 3 = 1750$, which is only seven eighths as long as with the original code. The reason this code does better is that we encode more frequent letters by shorter strings.

There are some uniquely decodable codes that are not prefix codes, but it can be shown that for any uniquely decodable code, there is a prefix code that compresses just as much. Hence we can restrict our attention to prefix codes.

So our task is: given an alphabet C with a frequency corresponding to each letter, find a binary prefix code that yields the shortest encoding given the frequencies. Such an optimal code is a Huffman Code.

An easy way to picture a binary prefix code for an alphabet C is by a binary tree where each letter of C is the label of a unique leaf. Then the codeword for a given letter c is found by taking a path from the root to the leaf c: if the path goes to the left child, then the next bit of the codeword is 0; if right child, then the next bit is 1. Since no labeled node is the ancestor of any other, the tree describes a prefix code. [Draw the tree for the example above.]

Here is the algorithm to construct a Huffman encoding tree bottom-up, given an alphabet $C = \{c_1, \ldots, c_n\}$, where each $c \in C$ has a integer frequency attribute $f[c] > 0$.

```
HuffmanTree(C)
    Let Q be a min priority queue (empty)
    Insert all letters c ∈ C into Q, keyed by frequency
    For i←1 to n − 1 do
        x ← ExtractMin(Q)
        y ← ExtractMin(Q)
        Form a new internal node z
        f[z]←f[x] + f[y]
        left[z]←x
        right[z]←y
        Insert z into Q
    // Return the root of the tree
    Return ExtractMin(Q)
```

The algorithm works by repeatedly merging a pair of nodes into a new node whose frequency is the combined frequencies of the original nodes. What is greedy about it? By merging two trees, we are essentially adding 1 to the lengths of all the codewords in the two trees (by virtue of the two edges from the new parent). Since we are making these codewords longer, we want their total frequency to be as low as possible. Hence, at each step we only merge the two available nodes with lowest possible frequency.

The algorithm above can be recast as a recursive algorithm. It may be easier to see how the correctness proof works for the recursive version:

```
HuffmanTree(C)
    if |C| = 1 then
        return the sole element of C
    let x, y be two elements of C
        with lowest freqency
    let z ∉ C be a letter
    f[z]←f[x] + f[y]
    C′←(C − {x, y}) ∪ {z}
    r ← HuffmanTree(C′)
    let T′ be the tree rooted at r
    make leaf z in T′ an internal node
        with children x and y
    return r
```

Lecture 19
Huffman Codes:
Proof of Correctness

To show that this algorithm produces an optimal tree, we first give an expression for the *cost* of a tree T, which is the total length of the encoding of a string whose letters occur with the given frequencies. The length of the codeword for a letter $c \in C$ is the depth of c in T. Thus the cost of

the tree is

$$B(T) = \sum_{c \in C} f[c] d_T(c),$$

where $d_T(c)$ is the depth of c in the tree T.

We fix an alphabet $C = \{c_1, \ldots, c_n\}$, where $n \geq 2$, and each c_i has a frequency $f[c_i]$. Our proof follows from two lemmas. The first says that the initial greedy merging step we take inside the for-loop is safe, in the sense that we won't miss an optimal tree starting this way. For our purposes, an *encoding tree for C* is a binary tree whose leaves are identified with the elements of C and each of whose internal nodes has two children. An encoding tree T for C is *optimal* if it $B(T)$ is minimum among the costs of all encoding trees for C.

Lemma 17 *Let T be any encoding tree for C, and let $x, y \in C$ have the two lowest frequencies of any letter in C. Then there is an encoding tree T' for C such that x and y are siblings on the deepest level of T', and further, $B(T') \leq B(T)$.*

Proof Let a and b be the two leftmost nodes on the deepest level of T. It is easy to see that a and b must have a common parent. Now if $\{x, y\} = \{a, b\}$, then we let $T' = T$ and we are done. Otherwise, suppose WLOG that $x \notin \{a, b\}$ and $a \notin \{x, y\}$. Then x is somewhere else in T, and $f[a] \geq f[x]$ by the choice of x and y. Let T' be the tree that results from swapping a with x in T. Then the only difference between T and T' is with the nodes x and a, where $d_T(x) = d_{T'}(a)$ and $d_T(a) = d_{T'}(x)$. Thus we have

$$\begin{aligned} B(T) - B(T') &= f[x]d_T(x) + f[a]d_T(a) \\ &\quad - f[x]d_{T'}(x) - f[a]d_{T'}(a) \\ &= f[x]d_T(x) + f[a]d_T(a) \\ &\quad - f[x]d_T(a) - f[a]d_T(x) \\ &= (f[a] - f[x])(d_T(a) - d_T(x)) \\ &\geq 0, \end{aligned}$$

because a is a node of maximum depth in T. Now if $b = y$, we're done. Otherwise, alter T' in the same manner as above by swapping b with y to get another tree with the same or smaller cost satisfying the lemma. □

The next lemma finishes the proof of correctness. It says that our sequence of reductions actually produces an optimal tree.

Lemma 18 *Let x and y be two letters in C with minimum frequency. Let $C' = (C - \{x, y\}) \cup \{z\}$ be the alphabet obtained from C by removing x and y and adding a new letter z (not already in C). Define the frequencies for letters of C' to be the same as for C except that $f[z] = f[x] + f[y]$. Suppose that T' is any optimal encoding tree for C', and let T be the encoding tree for C obtained from T by replacing leaf z with an internal node with children x and y. Then T is an optimal encoding tree for C.*

Proof First, we compare $B(T)$ with $B(T')$. The only difference is that z in T' is replaced with the parent of added nodes x and y in T, and so $d_T(x) = d_T(y) = d_{T'}(z) + 1$. All the other nodes are

the same in T as in T'. Thus,

$$\begin{aligned} B(T) &= B(T') - f[z]d_{T'}(z) + f[x]d_T(x) \\ &\quad + f[y]d_T(y) \\ &= B(T') - f[z]d_{T'}(z) + f[x](d_{T'}(z)+1) \\ &\quad + f[y](d_{T'}(z)+1) \\ &= B(T') + (f[x]+f[y]-f[z])d_{T'}(z) \\ &\quad + f[x] + f[y] \\ &= B(T') + f[x] + f[y]. \end{aligned}$$

Now we'll prove the lemma by contradiction. Assume that the hypotheses of the lemma but that T is not an optimal encoding tree for C. Then there is some encoding tree T'' for C with $B(T'') < B(T)$. We'll use T'' to construct an encoding tree T''' for C' with $B(T''') < B(T')$, which contradicts our assumption that T' was optimal.

By Lemma 17, we may assume WLOG that x and y are siblings in T''. Let T''' be the tree we get by removing x and y and replacing their parent with z. Note that T''' bears the exactly analogous relation to T'' as T' bears to T, namely, having z instead of a parent of x and y. Thus we can do the exact same calculation as we did above with $B(T)$ and $B(T')$, but this time with $B(T'')$ and $B(T''')$. This gives

$$B(T'') = B(T''') + f[x] + f[y].$$

Subtracting the second equation from the first, we get

$$B(T) - B(T'') = B(T') - B(T''').$$

The left-hand side is positive by assumption, so the right-hand side is also positive. But this means that $B(T''') < B(T')$, so T' is not optimal. □

It follows from this lemma that our algorithm is correct: the first iteration of the for-loop effectively reduces the construction of T to that of T'. Assuming inductively that the rest of the algorithm produces an optimal T', we know by the lemma that our T is optimal.

Lecture 20
Amortized Analysis

When performing a sequence of n operations on a data structure, we often are less concerned with the worst-case time taken by any single operation in the sequence, but rather the worst-case cost per operation averaged over the n operations.

Example: binary counter

Maintain an integer in binary. Operations: Reset (to zero), Increment, Display. Assume Reset is done once (at the beginning to initialize), followed by n increments, followed by Display. We will need $k = \lfloor \lg n \rfloor + 1$ bits for the counter. Incrementing is done in the usual way: adding 1 with possible carries. In the worst case, a single increment may take $\Theta(k)$ steps, e.g., when a carry propagates through all k bits. So we immediately get a bound of $O(nk) = O(n \lg n)$ for the total time taken for incrementing, for an average of $O(k)$ time per increment.

This is not tight, however, and we can do much better. The key observation is that the worst case does not happen very often. In fact, we require exactly i carries if and only if the result of the increment is an odd multiple of 2^i, since the carries will clear the i least significant bits, leaving the $(i+1)$st least significant bit equal to 1. This gives us a way to better compute the total time $T(n)$ taken by n increments (starting at zero). We'll assume that each carry takes unit time, and that the total cost of an increment other than carries is also unit time, so that the total cost of an increment is one more than the number of carries. We group the sum for $T(n)$ by the number i of carries required. Note that there are $O(n/2^i)$ many odd multiples of 2^i between 1 and n.

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lfloor \lg n \rfloor} (i+1)O(n/2^i) \\ &= O\left(n \sum_{i=0}^{\lfloor \lg n \rfloor} \frac{i+1}{2^i}\right) \\ &= O\left(n \sum_{i=0}^{\infty} \frac{i+1}{2^i}\right) \\ &= O(n), \end{aligned}$$

since the infinite sum converges.

Three methods for analyzing the amortized complexity: aggregate, accounting, and potential. The potential method subsumes the other two.

Example: stack with multipop

Amortized time for a sequence of n operations (starting with an empty stack) is $O(1)$ per operation, since at most n items are ever added to the stack, and each such item is "handled" for only $O(1)$ time. (Aggregate analysis.)

Example: dynamically resizing an table (array)

Assume Insert and Delete take unit time each (one dollar). If only Insert is performed, then a standard practice when the array becomes full is to allocate another array twice the size, and copy elements from the old array into the new. (Assume that the cost of the actual allocation is dominated by the copying costs, so we ignore it.) Thus the array will always be at least half full. We "charge" three units (three dollars) for each insertion. One dollar pays for the current insertion, one dollar pays for the next time the item is copied into a larger array (which we assume has unit cost), and the last dollar pays for another item's next copy into a larger array. Assuming an array currently of size $s \geq 2$ (s is a power of 2) containing exactly $n = s/2$ items. Then exactly n more items will be inserted before the array is resized. Each such additional item pays three dollars, one of which is spent immediately to insert the item, with the other two "put in the bank." After the n additional insertions, there are exactly $2n = s$ dollars in the bank, which is just enough to pay for the s items being copied from the smaller to the larger array. We withdraw all this money from the bank. (Accounting method.)

It is desirable to save space by allocating no more than $O(1)$ space per item, so if Delete operations are possible, we want to reduce the space allocated for the array when the number of items reaches 1/4 of the capacity of the array. (Why not 1/2?) We'll use this case to illustrate the potential method.

The Potential Method

We analyze the amortized complexity of a sequence of operations $a_1, \ldots, a_n$ on a data structure S. Let S_i be the state of the S after the ith operation a_i (S_0 is the initial state of S before a_1). For $0 \leq i \leq n$ we assign a real value $\Phi(S_i)$ (the *potential*), which depends on the state of S. Initially, $\Phi(S_0) = 0$, and $\Phi(S_i)$ will always be nonnegative. In analogy to the accounting method, the potential represents how much money the data structure has stored up to pay for future operations. If c_i is the actual cost of operation a_i, then we define the *amortized cost* of operation a_i to be

$$\widehat{c_i} = c_i + \Phi(S_i) - \Phi(S_{i-1}).$$

That is, $\widehat{c_i}$ is the actual cost, adjusted by the net change in the potential. Let T be the total actual time for operations $a_1, \ldots, a_n$. Since the potential starts at zero and remains nonnegative, we have

$$\begin{aligned} T &= \sum_{i=1}^{n} c_i \\ &\leq \sum_{i=1}^{n} c_i + \Phi(S_n) \\ &= \sum_{i=1}^{n} [c_i + \Phi(S_i) - \Phi(S_{i-1}))] \\ &= \sum_{i=1}^{n} \widehat{c_i}. \end{aligned}$$

Thus the total time is bounded from above by the total amortized time, and so any bound on the amortized time gives at least as good a bound on the actual time. We choose a potential function to make the worst-case amortized time of any operation as small as possible.

Back to Array Resizing

Let n be the current number of items in the array (between operations), and let s be the current size of the array. Set $\alpha = n/s$ (the load factor). Then it is always the case that $\alpha \geq 1/4$. Define the potential in this case as

$$\Phi = \begin{cases} 2n - s & \text{if } \alpha \geq 1/2, \\ s/2 - n & \text{if } \alpha < 1/2. \end{cases}$$

There are several cases to work through here (Exercise):

- $\alpha < 1/2$ after Insert,
- $\alpha < 1/2$ before Insert, and $\alpha \geq 1/2$ afterwards,
- $\alpha \geq 1/2$ before Insert, and $\alpha < 1$ afterwards,
- Insert makes $\alpha = 1$ so array needs expanding,
- $\alpha \geq 1/2$ after Delete,
- $\alpha \geq 1/2$ before Delete, but $\alpha < 1/2$ afterwards,
- $\alpha < 1/2$ before Delete, and $\alpha > 1/4$ afterwards,
- Delete makes $\alpha = 1/4$ so array needs contracting.

In all cases we see that the amortized time of any operation is always $O(1)$, so this is optimal.

Potential Method for Previous Examples

Multipop stack: Let $\Phi(S)$ be the number of items in S. Push only increases the potential by 1, so its amortized cost is 2. Pop and multipop both have amortized cost 0.

Binary counter: Let $\Phi(C)$ be the number of ones in counter C. Then each increment just changes the least significant 0 to 1, and each carry changes a 1 to 0, and thus is compensated for by a decrease in the potential. Thus increment has amortized cost $O(1)$.

When is amortized complexity not an appropriate measure of complexity? In real-time systems, especially critical systems, or when data structures are shared between many users (e.g., in a database), so that fair access is desired.

Lecture 21
Binomial Heaps
Disjoint Sets

Mergeable Heaps

A *mergeable heap* (min heap) supports all the usual heap operations: MakeHeap, Insert, Minimum, ExtractMin, as well as supporting Union, which takes two mergeable heaps and returns a single (mergeable) heap with all the items of the two heaps combined (the two original heaps are destroyed). Without Union, regular binary heaps work fine, but combining two binary heaps into one needs linear time. We'll see an implementation of mergeable heaps: *binomial heaps*, where all these operations, including Union, take time $O(\lg n)$ in the worst case. These heaps also support DecreaseKey and Delete in $O(\lg n)$ time.

Binomial Heaps

A *binomial tree B_k of height k* is an ordered tree whose shape is defined recursively in k as follows:

- B_0 is a single node.
- For $k > 0$, B_k is the tree that results from *linking* two binomial trees B_{k-1} of height $k-1$, so that the root of one becomes the leftmost child of the other.

[Draw a picture of B_0, B_1, B_2, and B_3.]

The following properties of B_k are verified by induction:

- B_k has exactly 2^k nodes.
- B_k has height k.
- There are exactly $\binom{k}{i}$ many nodes at depth i in B_k, for $0 \le i \le k$. (Induction on k and i.)
- The root of B_k has degree k, which greater than that of any other node.
- If the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$, then child i is the root of a subtree B_i.

It follows immediately that the maximum degree of any node in a binomial tree of size n is $\lg n$.

A *binomial heap H* is a list of binomial trees, whose nodes contain items, satisfying the following binomial heap properties:

- Each binomial tree in H obeys the usual min-heap property: the key of any nonroot node is at least the key of its parent. (The tree is *min-heap-ordered*.)
- The heights of successive trees in the list H are strictly monotone increasing. (In particular, no two trees in H have the same height.)

Suppose H has size $n > 0$ and consists of $\ell > 0$ many trees. Let $k_0 < k_1 < \cdots < k_{\ell-1}$ be the sequence of tree heights. Then, since $k_{\ell-1} \geq \ell - 1$, we have

$$n = \sum_{i=0}^{\ell-1} 2^{k_i} \geq 2^{\ell-1}.$$

This implies $\ell = O(\lg n)$. Conversely, any number $n > 0$ is uniquely expressible as a nonempty sum $\sum_{i=0}^{\ell-1} 2^{k_i}$ of increasing powers of two, thus the whole shape of H is determined exactly by its size alone.

We use the leftmost-child/right-sibling representation to implement binomial heaps. We use the right-sibling link to link the roots of the trees together (the *root list*, pointed to by $head[H]$). Each node x also contains the number of its children in $degree[x]$.

Making an Empty Binomial Heap

An empty binomial heap H is represented by $head[H]$ being null. This is what MakeHeap accomplishes, and it takes $O(1)$ time.

Finding the Minimum in a Binomial Heap

Traverse the list of tree roots, and return the minimum key found. This takes time linear in the number of trees, which is $O(\lg n)$.

Merging Two Binomial Heaps

Given binomial heaps H_1 and H_2, we traverse forward the root lists $head[H_1]$ and $head[H_2]$, merging the two lists by increasing tree height. If we encounter a tree B_k in H_1 and another tree B_k in H_2, then we perform a "carry" operation: we link the two together into a single B_{k+1} by making the root with bigger key the child of the root with smaller key. We then merge this new B_{k+1} (a one-item list) in with the two other lists. Since we never encounter more than three trees of the same size this way, it will work. This is the Union operation.

Since we do a constant amount of work on each tree, the time we take is linear in the total number of trees, which is logarithmic in the size of the newly created heap.

Inserting into a Binomial Heap

To insert x into H with n items, we first make a one-element heap H' containing x, then merge it with H. This takes time $O(\lg n)$.

Extracting a Minimum from a Binomial Heap

To remove a minimum element from H, we first find a minimum element, which will be the root of one of H's trees B_k. We remove B_k from the root list $head[H]$. Then we remove the root of B_k, leaving a list of exactly k children of heights $k-1, k-2, \ldots, 0$. We then *reverse* this list to make a binomial heap H'. Finally, we merge H' with H.

The total time for this is $O(\lg n)$: since $k = O(\lg n)$, we can reverse the list of children within this time, and finding a minimum and merging each can be done within this time.

Decreasing a Key in a Binomial Heap

After decreasing the key of some node in some tree B_k of H, we simply cascade the node up through B_k as necessary, just like we did with binary heaps. Since B_k has height $k = O(\lg n)$, the time taken is $O(\lg n)$.

Deleting a Node from a Binomial Heap

To delete an arbitrary node x in H, first perform DecreaseKey, decreasing x's key to $-\infty$. Then do an ExtractMin. The two actions combined take $O(\lg n)$ total time.

Question: How can we augment a binomial heap so that finding the minimum takes $O(1)$ time, without affecting the asymptotic times of the other operations? (This is mostly an academic exercise; there is not much practical reason to do this.)

Lecture 22
Disjoint Sets
Starting Graph Algorithms

Another useful data structure is one for maintaining a collection of pairwise disjoint nonempty sets $C = \{S_1, \ldots, S_k\}$. Each set S_i contains a unique distinguished element called its *representative*. Any element of S_i can serve as its representative, but there are never two or more representatives of S_i at the same time. The set S_i is identified by its representative.

The three supported operations are as follows:

MakeSet(x) creates a new set whose sole element is x. The representative of the new set is x (of course). To maintain disjointness with other sets, we require that x not belong to any other set.

Find(x) returns the unique set S_i containing x. Actually, it returns the representative of S_i.

Union(x,y) merges the set containing x with the set containing y into a single set (if the two sets were different). The two original sets are destroyed.

Note that we can test whether two elements x and y are in the same set, because this is equivalent to Find(x) and Find(y) being equal.

Applications

This data structure has many applications.

Example: connected components in a graph

Example: building a maze

Example: Unification pattern matching

We'll see others.

We often represent each set in the collection as an unordered, rooted tree of its elements, with the representative at the root and the other elements linked using parent pointers only. (This is the so-called disjoint set forest.)

We can implement Find(x) by following the parent pointers along the path from x to the root. We can implement Union(x, y) by first finding the roots of the corresponding trees (using Find(x) and Find(y)), then making one root a child of the other. The one remaining root becomes the representative of the combined set.

This basic approach works. If we are not careful, however, it may produce tall trees which could make Find(x) costly in the worst case. We can remedy this by following two additional heuristics:

Path-compression When performing Find(x), once the root is found, alter the parent pointers of all nodes on the path just traversed to point to the root directly. This requires keeping each node along the path in some data structure, like a stack, during traversal.

Union-by-rank When performing Union(x, y), choose more wisely which root to make a child of the other. Augment the data structure by maintaining some kind of extra information at each root (its *rank*) to help with the decision. Always make the root with smaller rank point to the root with larger rank, then update the rank of the remaining root.

There are different possibilities for union-by-rank. If we keep track of the height of each tree at its root (so rank equals height), then we would always make the root of the shorter tree the child of the root of the taller one. This won't increase the height unless the two heights are equal to begin with. Unfortunately, maintaining the height of a tree (e.g., updating it correctly after a Find) is too difficult and time-consuming for this approach to be useful. Instead, we define the rank as if it were the height, except that we don't try to alter it after a Find operation. Thus the rank is an upper bound on the height. When combining via Union two trees T_1 and T_2 with ranks r_1 and r_2 respectively, we increment the rank of the combined root if $r_1 = r_2$, and otherwise leave it alone.

Analysis

The time to perform m disjoint-set operations on n elements is known to be $O(m\alpha(n))$, where α is a monotone, *very* slowly growing (but still unbounded) function of n. For example, $\alpha(n)$ grows even more slowly than $\lg^* n$. So for all practical purposes, we have nearly constant amortized time per operation. We won't do the analysis here, but it is very clever, and you are welcome to read it from the book (I won't test on it).

Graph Algorithms

Let $G = (V, E)$ be a directed graph (digraph). $V = \{v_1, \ldots, v_n\}$ is the set of vertices of G, and $E \subseteq V \times V$ is the set of edges of G. If $(u, v) \in E$, then we say that v is *adjacent* to u, but not vice versa unless $(v, u) \in E$ (adjacency is not necessarily a symmetric relation).

There are two standard ways of representing G with a data structure.

Adjacency Matrix. We encode G by an $n \times n$ matrix A whose (i, j)th entry is

$$A[i, j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise,} \end{cases}$$

for $1 \le i, j \le n$.

Adjacency List. We maintain an array $V[1 \ldots n]$ of n linked lists, where $V[i]$ is (a reference to) a list of records representing edges leaving v_i (in no particular order). Each record in the list $V[i]$ records a different index j such that $(v_i, v_j) \in E$. (This is also known as the edge list representation.)

These data structures may be augmented with other information, such as edge weights, vertex weights, Boolean markers, et cetera.

We sometimes depict the directed edge (u, v) as $u \to v$ or as $v \leftarrow u$. We usually disallow self-loops in a graph, i.e., edges of the form $v \to v$.

We can use the same structures to represent undirected graphs by considering a undirected graph to be a special case of a digraph where each undirected edge $u \leftrightarrow v$ is represented by the pair of directed edges $u \to v$ and $u \leftarrow v$. Thus the adjacency matrix for an undirected graph is symmetric ($A[i, j] = A[j, i]$ for all i, j).

Note that $|E| \leq |V|^2$. If $|E| = \Omega(|V|^2)$ then we say that the graph is "dense." Likewise, if $|E| = o(|V|^2)$, then the graph is "sparse." (We are abusing terminology here. These properties really only apply to infinite classes of graphs rather than individual graphs.) The adjacency matrix for G has size $\Theta(|V|^2)$, and the adjacency list for G has size $\Theta(|V| + |E|)$. If G is dense, then both representations above are roughly equivalent in size. If G is sparse, however, the adjacency list representation is more compact, and so we generally prefer it. We'll assume that all our input graphs are given in the adjacency list representation, and hence the *size* of G will be the size of this representation of G, i.e., $\Theta(|V| + |E|)$.

If G is dense, the adjacency matrix representation may be more useful if we want to find out quickly whether two given vertices are connected by an edge. It's easy to convert between the two representations.

Exercise: The edge list representation is convenient for following edges forward, but inconvenient for following edges backward (which we occasionally wish to do). For any digraph $G = (V, E)$, we define G^T to be the graph (V, E'), where $E' = \{(v, u) \mid (u, v) \in E\}$. Thus G^T is the same as G but with all edges pointing in the opposite direction. Traversing an edge backward in G is thus the same as traversing the corresponding edge forward in G^T. Describe an algorithm that takes an edge list representation of any digraph G and produces an edge list representation of G^T. Your algorithm should run in *linear time* (linear in the size of the input representation). (We use the notation G^T because the adjacency matrix of G^T is the transpose of that of G. Some people use G^R or G^r instead.)

Lecture 23
Graph Search

Probably the most fundamentally useful graph algorithm is *search*, which systematically visits all vertices or edges in a graph or part of a graph. There are different kinds of graph search, but all common types of graph search are special cases of a generic search algorithm that we now describe. (This is not in the book!)

During the search, each vertex will be one of three colors: white, grey, or black. White vertices are those that have not been found yet; grey vertices are those that have been discovered but not completely processed; black vertices have been completely processed and will not be processed further. We use some collection B (called the "box") to hold the grey vertices. The data structure used for B can vary depending on the type of search. The only operations we require B to support are

Make(B) (re-)initializes B to be empty,

Empty(B) tests whether B is empty,

Insert(x, B) inserts a vertex x into B, and

Delete(B) removes and returns some element x from B (assuming B is not empty).

Possible box "types" include stack, queue, or priority queue.

We assume that each vertex has a *color* attribute that is either white, grey, or black.

GenericSearch takes a graph $G = (V, E)$ as input, and searches all white vertices of G.

```
GenericSearch(G)
    for each vertex v of G do
        if color[v] = white then
            GenericSearchAt(G, v)
```

GenericSearchAt takes a graph $G = (V, E)$ and a vertex $v \in V$ as input, and searches that portion of the graph that is reachable from v through white vertices only. We use three "visitation" subroutines: Start, Update, and Finish. These three routines will vary depending on the type of search. They may also share a persistent state (e.g., they build some structure incrementally over several calls). We assume that none of these routines alters the *color* attribute of any vertex; we do that explicitly in GenericSearchAt.

```
GenericSearchAt(G, v)
// Assumes color[v] = white
    // B is a local box
    Make(B)
    Start(v; nil)
    color[v]←grey
    Insert(v, B)
    Update(v; nil, B)
    repeat
        u ← Delete(B)
        Finish(u)
        color[u]←black
        for each w adjacent to u do
            if color[w] = white then
                // we'll say that u discovers w here
                Start(w; u)
                color[w]←grey
                Insert(w, B)
            if color[w] = grey then
                Update(w; u, B)
    until Empty(B)
```

GenericSearchAt finds new vertices by their being adjacent to old ones. The subroutine Start is called on a vertex when it is first found and it enters the box (its color changing from white to grey). Once in the box, the vertex will be Update'd one or more times, once for each edge leading to it from a black vertex.

The time taken by GenericSearchAt is dominated by the calls to Insert, Delete, Start, Update, and Finish. The following table gives the maximum number of times each subroutine is called:

Routine	**# of calls**
Insert	$\|V\|$
Delete	$\|V\|$
Start	$\|V\|$
Update	$\|E\| + 1$
Finish	$\|V\|$

Breadth-First Search (BFS)

This is the type of search that results from implementing the box B as a (simple) queue. BFS finds new vertices in order of increasing (unweighted) distance from v. It is used, for example, to find the unweighted distance from v to any node. (The unweighted length of a path is the number of edges along the path; the unweighted distance from node u to node v is the minimum unweighted length of a path from u to v, or ∞ if there is no path.)

Depth-First Search (DFS)

This type of search results from implementing B as a stack. DFS goes out as far as possible along a path from v before backtracking to another path.

Search Trees

A useful structure that can be produced from a search from a vertex v of G is a *search tree*. A search tree is an unordered, rooted tree (with root v), on some of the vertices of G, whose parent-to-child edges form a subgraph of G. A vertex is added to the tree as a new leaf when it is discovered, with parent the vertex that discovers it. Assuming that each vertex has additional attributes *parent*, *leftmost*, and *rightsibling*, we build a search tree by including the following code in Start$(u; w)$:

$parent[u] \leftarrow w$
$leftmost[u] \leftarrow nil$
$rightsibling[u] \leftarrow leftmost[w]$
$leftmost[w] \leftarrow u$

This code sets u to be a leaf with parent w, and adds u onto w's list of children.

In a breadth-first search tree, the path from the root to any vertex in the tree is a shortest path (unweighted).

Depth-first search trees are useful for finding, among other things, strongly connected components in a digraph and articulation points in an undirected graph. (A digraph is *strongly connected* if for every ordered pair (u, v) of vertices there is a directed path from u to v. In a general digraph G, we'll say that two vertices u and v of G are *equivalent* ($u \equiv v$) if there are directed paths from u to v and from v to u. The relation $\equiv$ is clearly an equivalence relation, and the equivalence classes are the strongly connected components of G. For an undirected, connected graph G, an articulation point is a vertex whose removal disconnects the remaining graph.)

Dijkstra's Algorithm for Single-Source Shortest (Weighted) Path

We are given a digraph $G = (V, E)$, where each edge (u, v) has a real-valued attribute $c[u, v]$ called the *cost*, or *distance*, of the edge (u, v). (Numerical attributes such as these are called *edge weights*. In the adjacency list representation, edge weights are stored in the linked list nodes.) We are also given a source vertex $s \in V$. We would like to find the *distance* and a shortest (directed) path (if it exists) from s to each other vertex, where the length of a path is the sum of the costs of the edges along the path. BFS solved this in the case where all the edge costs are equal to one, but here they could be arbitrary (except that we forbid cycles of negative length).

For example, we want to find the shortest routes from the warehouse to each of the retail stores.

A beautiful algorithm of Dijkstra solves this problem in the case where all edge costs are nonnegative. It is a special case of GenericSearchAt(G, s) above, where

- Each vertex v has two attributes:
  - $d[v]$ is a real number that will eventually hold the (shortest) distance from s to v. If v is not reachable from s, then we make the distance ∞ by convention.
  - $b[v]$ will eventually point to the predecessor of v along a shortest path from s to v. Following these fields backwards gives a shortest path from s to v.
- Initially, $d[s] = 0$ and $d[v] = \infty$ for all $v \neq s$.
- Initially, $b[u] = \mathit{nil}$ for all $u \in V$.
- The box B is a min-heap, with items (vertices) keyed by their d attributes. Insert and DeleteMin are the relevant operations. We'll also use DecreaseKey (see Update, below).
- Start and Finish are both no-ops.
- The Update$(v; u, B)$ procedure checks if new evidence has been found to decrease $d[v]$ based on v being adjacent to u. It is defined as follows:

  Update$(v; u, B)$
  if $d[u] + c[u, v] < d[v]$ then
  // This implicitly sets $d[v] \leftarrow d[u] + c[u, v]$
  DecreaseKey$(B, v, d[u] + c[u, v])$
  $b[v] \leftarrow u$

  We don't bother calling this routine initially on s. Note that d-values can only decrease; they never increase.

The correctness of the algorithm all hinges on the fact that, when a vertex v is removed from B, its d and b attributes are correct. We prove this by induction on the number of times DeleteMin(B) has been called so far. Here is the idea. First, it is clear that we never set the d attribute of any vertex x to a finite value unless we actually have evidence that there is a path from s to x with at most that length. This is easily shown by induction. So now we only need to show now that $d[v]$ is not too large when v leaves B. Suppose a vertex v is just about to be removed from B. Then $d[v]$ is a minimum key in B. Suppose there is a path p from s to v of length strictly less than $d[v]$. Following p backwards from v, there must be a point where we first go from a nonblack vertex x to a black vertex y (since v is grey but s is black). We now know the following are true at this point in time:

- x is grey (and thus x is in B). Since y is black and x is adjacent to y, x is discovered, so it cannot be white.
- $d[y]$ is at most the length of that part of p that goes from s to y. Since y is black, it was already removed from B previously, so by the inductive hypothesis, $d[y]$ is correct, i.e., it is the length of a shortest path from s to y, so $d[y]$ certainly can be no more than the length along p from s to y.
- $d[x]$ is at most the length of that part of p that goes from s to x. Since y is black, at some point the edge (y, x) was previously "traversed," i.e., Update$(x; y, B)$ was called. When this happened $d[x]$ was set to $d[y] + c[y, x]$ (if it was larger). But $d[y] + c[y, x]$ is exactly the length of that part of p that runs from s to x.
- $d[x] < d[v]$. By assumption $d[v]$ is strictly greater than the length of p, which in turn is at least the length along p from s to x (here we need the fact that there are no negative edge costs).

This last item is a contradiction, because $d[v]$ has to be the minimum key in B. Thus no such shorter path p can exist.

To show that the b-attributes are correct, we observe (by induction again), that at any time in the algorithm and any vertex v, if $b[v] \neq nil$ then $b[v]$ is the predecessor to v along a path of length $d[v]$ from s to v.

The worst-case running time for Dijkstra's algorithm is $\Theta((|V| + |E|) \lg |V|)$ if we implement B as a binary heap. The reason for this is that B may have size as much as (but no more than) $|V|$, and heap operations are logarithmic time.

Lecture 24
Minimum Spanning Trees

You have a bunch of nodes (computers or other devices) and you want to connect them all to a common LAN in the cheapest possible way by laying physical links between pairs of nodes. The only restriction is that there be a path in the network connecting any pair of nodes (the path may go through several other nodes). If $c[i, j]$ is the cost of stringing a link between nodes i and j, then the total cost of setting up the network is the sum of the costs of all the links strung.

One obvious rule to follow (assuming all the $c[i, j]$ are positive) is that there should be no cycles in the network, for if there were a cycle p, removing one link from p would give a cheaper network with the same connectivity. So the cheapest network will be a tree spanning all the nodes.

We'll go over two greedy algorithms that each find a minimum spanning tree in a graph. The proofs of correctness for both algorithms are similar, relying on the same lemma.

Definition 19 *Let G be an undirected, connected graph. A* spanning tree *for G is a collection of edges of G that forms a tree on all the vertices of G. If G has edge weights, then the* weight *of a spanning tree T is the sum of the weights of the edges in T. A* minimum spanning tree *for G is a spanning tree whose weight is minimum (among all possible spanning trees).*

It's a good exercise to prove the following:

Fact 20 *Let G be an undirected graph with n vertices.*

- *If G is connected, then G has at least $n-1$ edges.*
- *If G has at least n edges, then G has a cycle.*
- *If G is a tree (i.e., connected and acyclic), then G has exactly $n-1$ edges. (This follows from the two previous facts.)*

Edge weights (costs) for a graph $G = (V, E)$ are assumed given by a function $w : E \to \mathbb{R}$.

Kruskal's Algorithm

This algorithm uses a system of disjoint sets of edges.

```
KruskalMST(G, w)
    T←∅
    for each v ∈ V do
        MakeSet(v)
    sort edges of E into increasing order by w
    for each (u, v) ∈ E in order, do
        // Invariant (proved below):
        // T is a subset of some MST for G
        if FindSet(u) ≠ FindSet(v) then
            T←T ∪ {(u, v)}
            Union(u, v)
    // T is an MST for G (proved below)
    return T
```

During the execution of this algorithm, T will be a forest of trees unconnected to each other, starting with $|V|$ many empty trees (single isolated vertices with no edges). We repeatedly add to T the lightest possible edge that joins two unconnected trees together.

If G is not connected, then no spanning tree exists. The algorithm can detect this: G is connected if and only if exactly $|V| - 1$ Union operations are performed (if less, then T will be a forest of at least two trees).

Prim's Algorithm

This is a special case of generic search. It is similar to Dijkstra's algorithm, but with one crucial difference in the Update routine. Each vertex has a real-valued attribute d and a vertex-valued attribute π. The MST is built as a rooted tree whose root can be any vertex. The final π-value of each vertex will be its parent pointer.

```
PrimMST(G, w)
    for each vertex v do
        d[v]←∞
        π[v]←nil
    let s be any vertex of G
    d[s]←0
    GenericSearchAt(G, s)
    // MST is encoded in π-values
    T←∅
    for each vertex v do
```

if $\pi[v] \neq nil$ do
$T \leftarrow T \cup \{(v, \pi[v])\}$
return T

The subroutine GenericSearchAt(G, s) is implemented as follows:

- The box B is a min-heap with items keyed by their d-values.
- Start and Finish are no-ops.
- Update$(v; u, B)$
  if $w[u, v] < d[v]$ then
  $\pi[v] \leftarrow u$
  $d[v] \leftarrow w[u, v]$

During the generic search phase, a single tree is grown, starting with the root s, by repeatedly adding new vertices (that are not already part of the tree) as leaves. The new leaf is chosen each time so as to add the lightest possible edge to the tree.

How can this algorithm be used to test whether G is connected?

Correctness

Each algorithm is greedy in that the lightest edge satisfying some criterion is added to the tree in each step. The correctness of both approaches will follow from Lemma 23, below, which says that when building an MST, by accumulating edges, certain edges are "safe" to add.

Definition 21 *Let $G = (V, E)$ be an undirected graph. A* cut *in G is a partition of V into two sets S and $V - S$. Such a cut is denoted by $(S, V - S)$.*

Definition 22 *Let $(S, V - S)$ be a cut in G. An edge (u, v)* crosses *the cut $(S, V - S)$ if one of the endpoints u and v is in S and the other is in $V - S$. If A is a set of edges, then we say that the cut* respects *A if no member of A crosses the cut.*

Lemma 23 *Let $G = (V, E)$ be a connected, undirected graph with edge weights w. Suppose that A is a subset of some minimum spanning tree for G, and let $(S, V - S)$ be any cut that respects A. If (u, v) is an edge crossing $(S, V - S)$ of minimal weight, then $A \cup \{(u, v)\}$ is also a subset of some minimum spanning tree for G. (We say that (u, v) is a* safe *edge to add to A.)*

Proof Let T be some MST containing A. If $(u, v) \in T$, we are done, so assume otherwise. Adding (u, v) to T yields a set $U = T \cup \{(u, v)\}$ of $|V|$ many edges, which thus must have a cycle. Further, U must have a cycle c containing (u, v), since T itself is acyclic. Since (u, v) crosses $(S, V - S)$, there must be some other edge on c besides (u, v), say (x, y), that also crosses $(S, V - S)$. Then $(x, y) \in T$, and by the assumption that (u, v) has minimum weight, we have $w[x, y] \geq w[u, v]$. Now let $T' = U - \{(x, y)\}$. T' had exactly $|V| - 1$ edges and is connected (any path in T that goes through (x, y) can be rerouted through (u, v) to be a path in T'). Thus T' is a spanning tree, and

$$w(T') = w(T) + w[u, v] - w[x, y] \leq w(T).$$

Since T is an MST, we must have $w(T') = w(T)$ (and so also $w[u, v] = w[x, y]$) and T' is also an MST. Further, $A \cup \{(u, v)\} \subseteq T'$, and so $A \cup \{(u, v)\}$ is contained on some MST. □

We now see how Lemma 23 implies the correctness of Kruskal's and Prim's algorithms. In each case, we only add safe edges.

In Kruskal's algorithm, we accumulate edges into T, maintaining the invariant that T is always a subset of some MST. We only need to show that when an edge (u, v) is added to T, there is some cut respecting T such that (u, v) is a minimum weight ("light") edge crossing the cut. At any time in the algorithm and any vertex w, we let S_w be the set in the disjoint set system that currently contains vertex w. When edge (u, v) is added to T, Union(u, v) is called, which joins the two previously disjoint sets S_u and S_v. Consider the cut $(S_u, V - S_u)$, which clearly respects T before (u, v) is added. Any edge that crosses this cut has one vertex in S_u and the other in some $S_w \neq S_u$, and so has not previously been added to T. Since we take the edges in increasing order by weight, it must be that (u, v) is a minimum weight edge crossing the cut. Thus it is safe to add to T, and the loop invariant is maintained.

In Prim's algorithm, at any time during execution, let

$$T = \{(v, \pi[v]) \mid v \text{ is black and not the root}\}.$$

T is initially empty, and any edge $(v, \pi[v])$ enters T when v is removed from B and turns black. (T is not actually maintained by the algorithm during the search, but we could easily alter the search to maintain T.) (Note that $\pi[w]$ is always black from the time a vertex w is first updated.) Immediately before v leaves T, let S be the set vertices that are currently black, and consider the cut $(S, V - S)$, which clearly respects T. Since $\pi[v]$ is black and v is grey, $(v, \pi[v])$ crosses the cut $(S, V - S)$.

Claim 24 *$(v, \pi[v])$ is a minimum weight edge crossing the cut $(S, V - S)$, and is thus safe to add to T.*

Proof Let (x, y) be any edge crossing $(S, V - S)$. We show that $w[x, y] \geq w[v, \pi[v]]$. Immediately before v is removed from B, one of the vertices x or y is black (say x), and the other (say y) is grey, since it is adjacent to x but not black. So y is in the box at this time. We have $d[y] = w[y, \pi[y]] \leq w[x, y]$, because the algorithm guarantees that $d[y]$ is the minimum weight of any edge connecting a black node with y. We also have $d[v] = w[v, \pi[v]]$. But since v is about to be removed from B (a min-heap), we must have $d[v] \leq d[y]$, which proves the claim. □

Prim's algorithm is thus correct, because only safe edges are added to T during the search phase.

Lecture 25
NP-Completeness

This is the most important subject that most students don't learn well enough.

- Given a set of S integers, is there a way to partition S into two subsets whose sums are equal?
- Given a graph G, is there a path in G that goes though each vertex exactly once?
- Given a graph G and an integer K, is there a set C of no more than K vertices such that every edge in G is incident to at least one vertex in C?
- Given a Boolean formula φ, is there a truth-setting of the variables that makes φ true?

- Given a graph G and an integer K, does G have a complete subgraph of size at least K?

No algorithms are known for any of these problems that run in less than exponential time (essentially by exhaustive search).

BUT, a fast algorithm for any one of them will immediately give fast algorithms for the rest of them.

All these problems, and many others, are NP-complete.

The theory of NP-completeness is the best tool available to show that various interesting problems are (most likely) *inherently difficult*.

[So far, no one has been able to prove mathematically that NP-complete problems *cannot* be solved by fast algorithms, but this hypothesis is supported by a huge amount of empirical evidence, namely, the failure of *anybody* to find a fast algorithm for *any* NP-complete problem despite intense and prolonged effort.

So if your boss asks you to find a fast algorithm for a problem and you cannot find one, you may be able to show your boss that the problem is NP-complete, and hence equivalent to the problems above, which the smartest minds in the field have failed to crack.

At least your boss would know that she won't do any better by firing you and hiring someone else.]

Decision Problems

We restrict our attention to *decision* (i.e., yes/no) problems for convenience.

Definition 25 *A* decision problem *is specified by two ingredients:*

1. *a description of an* instance *of the problem (always a finitely representable object), and*
2. *a yes-no question regarding the instance.*

So a decision problem is a set of instances, partitioned into yes- and no-instances.

All the questions above are decision problems. When stating a decision problem, we name it, then explicitly give its two ingredients. For example,

> VERTEX COVER
> Instance: an undirected graph G and an integer K.
> Question: is there are set of vertices C of size at most K such that every edge in G is incident to a vertex in C?

All instances of a decision problem are either yes-instances or no-instances, depending on the answer to the corresponding question.

[We can apply these techniques to other kinds of problems, e.g., search problems, if we wanted. Often, a fast algorithm for a decision problem can be used to get a fast algorithm for a related search problem. For example, suppose we had an algorithm for VERTEX COVER, above, and we wanted an algorithm to find a vertex cover of maximum size in a graph. We can first call VERTEX COVER repeatedly with different K values to determine first the size of a maximum cover. Then we can find an actual cover of this size by repeatedly calling VERTEX COVER on graphs obtained by removing successive vertices from the original graph.]

P and NP

We say that an algorithm A *solves* a decision problem L if, given any instance of L as input, A outputs the correct answer to the corresponding question.

Definition 26 *We define* **P** *to be the class of all decision problems that are solvable in polynomial time. That is,* **P** *is the class of all decision problems* Π *for which there exists a constant* k *and an algorithm that solves* Π *and runs in time* $O(n^k)$*, where* n *is the size (in bits) of the input.*

Thus **P** is the class of all decision problems that are "easily decidable." ("easy" = polynomial time; we don't need any finer granularity)

[Without loss of generality, we will assume that all algorithms must read their input sequentially, as if from a file on disk, say. Likewise, all outputs must be written sequentially (e.g., to a disk file). Thus reading input and writing output take time proportional to the size of each. This will simplify much of the discussion below.]

A decision problem is in the class **NP** if all its yes-instances can be easily verified, given the right extra information.

For example, if a graph G does have a vertex cover C of size $\leq K$, this fact can be verified easily if the actual set C is presented as extra information (we simply check that each edge in G is incident to a vertex in C).

Such extra information is called a *proof* or *witness*.

Definition 27 **NP** *is the class of all decision problems* Π *for which there exists an algorithm* A *that behaves as follows for all instances* x *of* Π*:*

- x *is a yes-instance of* Π *if and only if there is a* y *such that* A *outputs "yes" on input* (x, y)*.*
- A *runs in time polynomial in the length of* x*.*

Such a y (when it exists) is a witness, and A is the algorithm that verifies, using the witness, that x is a yes-instance of Π. In the case of VERTEX COVER, x encodes a graph G and integer K, and y (if it exists) would encode a vertex cover for G of size at most K.

Since A must stop within time $O(n^k)$ for some constant k, it can only read the first $O(n^k)$ bits of y, where n is the length of x. Thus we can limit the size of y to be polynomial in n.

All the problems listed above are in **NP**. Also, it is clear that $\mathbf{P} \subseteq \mathbf{NP}$ (for a problem $\Pi \in \mathbf{P}$, a verifying algorithm could ignore any extra proof and just decides whether the input is a yes- or no-instance in polynomial time; hence $\Pi \in \mathbf{NP}$).

Reductions

We want to compare decision problems by their difficulty.

Definition 28 *Given two decision problems* Π_1 *and* Π_2*, we say that* Π_1 polynomially reduces to Π_2 *(*$\Pi_1 \leq^p \Pi_2$*) if there is a function* f *such that*

- f *maps each instance of* Π_1 *to an instance of* Π_2*,*
- f *can be computed in polynomial time, and*
- *for each instance* x *of* Π_1*,*

  > x *is a yes-instance of* Π_1 *iff* $f(x)$ *is a yes-instance of* Π_2 *(and thus* x *is a no-instance of* Π_1 *iff* $f(x)$ *is a no-instance of* Π_2*).*

f *is called a* polynomial reduction *from* Π_1 *to* Π_2*.*

This captures the notion that Π_1 is "no harder than" Π_2, or, Π_2 is "at least as hard as" Π_1. The $\leq^p$ relation is reflexive and transitive.

The intuition that $\leq^p$ at least partially orders problems by difficulty is supported by the following theorem.

Theorem 29 (Pretty easy) *Suppose* $\Pi_1 \leq^p \Pi_2$. *Then,*

- *if* $\Pi_2 \in \mathbf{P}$ *then* $\Pi_1 \in \mathbf{P}$, *and*
- *if* $\Pi_2 \in \mathbf{NP}$ *then* $\Pi_1 \in \mathbf{NP}$.

Proof Fix a polynomial reduction f from Π_1 to Π_2, running in time $O(n^k)$ for some k.

First, suppose $\Pi_2 \in \mathbf{P}$, decided by a deterministic decision algorithm A running in time $O(n^\ell)$ for some ℓ. Consider the following decision algorithm B for Π_1:

On input x, an instance of Π_1:
$z \leftarrow f(x)$
run A on input z and output the result

Since f is a reduction, z is an instance of Π_2 with the same answer as x in Π_1, so B correctly decides Π_1. Further, B runs in time polynomial in $n = |x|$: it first computes $z = f(x)$ (requiring time $O(n^k)$), which can have length no more than $m = O(n^k)$ (f does not have time to produce a bigger output); the call to A thus takes time $O(m^\ell) = O((n^k)^\ell) = O(n^{k\ell})$; thus the total time for B is polynomial in n. So B decides Π_1 in polynomial time, and so $\Pi_1 \in \mathbf{P}$.

Second, suppose $\Pi_2 \in \mathbf{NP}$, with a yes-instance verifier A running in time $O(n^\ell)$. Consider the following algorithm B to verify yes-instances of Π_1:

On input (x, y), where x is an instance of Π_1:
$z \leftarrow f(x)$
run A on input (z, y) and output the result

B runs in time polynomial in $n = |x|$ for reasons similar to those above. (Note that we can assume that $|y| = O(|z|^\ell) = O(n^{k\ell})$, since A does not have enough time to read more than this.) For correctness, we observe that x is a yes-instance of Π_1 iff z is a yes-instance of Π_2, iff there is a y such that A accepts (z, y), iff there is a y such that B accepts (x, y). Thus $\Pi_1 \in \mathbf{NP}$. □

Definition 30 *Two decision problems* Π_1 *and* Π_2 *are* polynomially equivalent *(*$\Pi_1 \equiv^p \Pi_2$*) if both* $\Pi_1 \leq^p \Pi_2$ *and* $\Pi_2 \leq^p \Pi_1$.

NP-Hardness and NP-Completeness

Definition 31 *A decision problem* Π *is* **NP**-hard *if, for every problem* $\Pi' \in \mathbf{NP}$, *we have* $\Pi' \leq^p \Pi$.

Thus a problem is **NP**-hard iff it is at least as hard as any problem in **NP**.

Definition 32 *A decision problem is* **NP**-complete *if it is in* **NP** *and it is* **NP***-hard.*

Theorem 33 (Easy) *Any two* **NP***-complete problems are polynomially equivalent.*

Proof If Π_1 and Π_2 are NP-complete, then in particular, $\Pi_1 \in \mathbf{NP}$, and everything in **NP** reduces to Π_2. Thus $\Pi_1 \leq^p \Pi_2$. Likewise, $\Pi_2 \leq^p \Pi_1$. □

All the problems listed above are **NP**-complete, and hence polynomially equivalent.

Lecture 26
NP-Completeness, Continued

The Standard Technique

The standard technique that we will use for showing that a problem Π is NP-complete takes two steps:

1. Show that $\Pi \in \mathbf{NP}$. This is usually obvious.
2. Find a polynomial reduction from Π' to Π for some known NP-complete problem Π'.

Since all NP problems are reducible to Π', and since the $\leq^p$ relation is transitive, it follows that all NP problems are reducible to Π, and thus Π is NP-complete.

Obviously, the first natural NP-complete problem could not be proved NP-complete using this method. The first such problem was

SATISFIABILITY (SAT)
Instance: A Boolean formula φ.
Question: Is φ satisfiable, i.e., is there a setting (truth assignment) of the Boolean variables of φ that makes φ true?

SAT is clearly in NP. Given a satisfiable formula φ, an easily verifiable proof that φ is satisfiable is a satisfying truth assignment of the variables. Given such an assignment, we simply compute the corresponding truth value of φ in the standard bottom-up way and check that φ is indeed true under the assignment.

In the 1970s, Steve Cook (Waterloo, Ontario, Canada) and Leonid Levin (then in the USSR, now at Boston U.) independently came up with an ingenious proof that SAT is NP-hard. They used the Turing machine model to describe algorithms. I prove this theorem when I teach CSCE 551. [Note: Garey & Johnson call this theorem Cook's Theorem. Levin's work was unknown to them at the time.]

Theorem 34 (Cook, Levin) *SAT is NP-complete.*

The Cook-Levin Theorem provides the starting point we need to use our technique. By now, there are hundreds (if not thousands) of known NP-complete problems to start from, and there is much variety (computer science, operations research, scheduling, game playing, etc.). Proving a problem to be NP-complete adds to this list, making it easier for further proofs, etc.

CNF-SAT

Cook and Levin actually showed that the following restriction of SAT is NP-complete:

CNF-SAT
Instance: A Boolean formula φ in conjunctive normal form (CNF).
Question: Is φ satisfiable?

A formula is in CNF if it is a conjunction

$$C_1 \wedge \cdots \wedge C_n$$

of *clauses* C_i, where a clause is defined as a disjunction

$$(\ell_1 \vee \cdots \vee \ell_m)$$

of *literals* ℓ_j, and where a literal is defined as either a Boolean variable (e.g., x) or the negation of a Boolean variable (e.g., $\neg x$, also written $\overline{x}$). Note here that $\wedge$ means AND, and $\vee$ means OR.

Thus a truth assignment satisfies a CNF formula if and only if it satisfies every clause, where a clause is satisfied if and only if at least one literal in the clause is true.

CNF-SAT may appear easier than SAT at first, since we only need to worry about formulas in CNF instead of arbitrary formulas. But CNF-SAT is NP-complete, so it is polynomially equivalent to SAT. We will take as given that CNF-SAT is NP-complete.

3-SAT

We can restrict CNF-SAT even further:

> 3-SAT
> Instance: A Boolean formula φ in CNF where each clause in φ has exactly three literals.
> Question: Is φ satisfiable?

3-SAT is NP-complete, and we can prove this using our standard technique by finding a polynomial reduction from CNF-SAT to 3-SAT.

[Interesting fact: If we restrict φ to have at most two literals per clause, the resulting problem, 2-SAT, can be solved in polynomial time.]

Here is how the reduction will work. Given an instance $\varphi = C_1 \wedge \cdots \wedge C_n$ of CNF-SAT, where the C_i are clauses, we will replace each clause $C = (\ell_1 \vee \cdots \vee \ell_m)$ of φ with a conjunction of one or more 3-literal clauses, depending on the value of m. We take the conjunction of all the resulting clauses as our output formula φ', an instance of 3-SAT. We must be sure that φ is satisfiable if and only if φ' is satisfiable.

If $m = 3$, then C already has three literals, so we leave it alone.

If $m = 2$ (so $C = (\ell_1 \vee \ell_2)$), then let y be a fresh variable (*fresh* means that y does not occur anywhere else), and replace C by the two clauses

$$(\ell_1 \vee \ell_2 \vee y) \wedge (\ell_1 \vee \ell_2 \vee \overline{y}).$$

Notice that any truth assignment that satisfies C makes at least one of ℓ_1 and ℓ_2 true, and so can be extended to a truth assignment (by giving any truth value for y) that satisfies both replacement clauses, above. Conversely any truth assignment that satisfies both clauses above must also satisfy C: if the assignment did not satisfy C, then ℓ_1 and ℓ_2 would both be false under the assignment, so no matter how y was set, one of the clause above must have been false.

If $m = 1$ (so $C = (\ell_1)$), let y_1 and y_2 be fresh variables. Replace C by the four clauses

$$(\ell_1 \vee y_1 \vee y_2) \wedge (\ell_1 \vee y_1 \vee \overline{y_2}) \wedge (\ell_1 \vee \overline{y_1} \vee y_2) \wedge (\ell_1 \vee \overline{y_1} \vee \overline{y_2}).$$

Again, any truth assignment satisfying C makes ℓ_1 true, and thus satisfies all four replacement clauses. Conversely, any truth assignment that satisfies all four clauses above must make ℓ_1 true, and thus satisfy C.

If $m > 3$, then using fresh variables $y_1, \ldots, y_{m-3}$, we replace C with

$$(\ell_1 \vee \ell_2 \vee y_1) \wedge (\overline{y_1} \vee \ell_3 \vee y_2) \wedge$$
$$\cdots \wedge (\overline{y_{i-2}} \vee \ell_i \vee y_{i-1}) \wedge \cdots$$
$$\wedge (\overline{y_{m-4}} \vee \ell_{m-2} \vee y_{m-3}) \wedge (\overline{y_{m-3}} \vee \ell_{m-1} \vee \ell_m).$$

For example, $(\ell_1 \vee \cdots \vee \ell_5)$ is replaced with

$$(\ell_1 \vee \ell_2 \vee y_1) \wedge (\overline{y_1} \vee \ell_3 \vee y_2) \wedge (\overline{y_2} \vee \ell_4 \vee \ell_5).$$

Suppose some truth assignment satisfies C. Then it makes some literal ℓ_i of C true. We can extend this assignment to satisfy all the replacement clauses simultaneously by making all the y's to the left of ℓ_i true and all the y's to the right of ℓ_i false. That is, we set y_j to true for all $j \leq i-2$ and we make y_j false for all $j \geq i-1$. The true y's satisfy all clauses to the left of the one containing ℓ_i, and the false y's satisfy all clauses to the right. ℓ_i alone satisfies the clause containing it.

Conversely, it is not too hard to see that the only way a truth assignment can satisfy all the replacement clauses simultaneously is by making at least one of the ℓ_i true. (Consider three exhaustive cases: y_1 is false; y_{m-3} is true; y_{i-2} is true and y_{i-1} is false, for some $3 \leq i \leq m-2$.) Thus, this assignment also satisfies C.

Now let φ' be the conjunction of all the replacement clauses described above, for all the clauses of φ. Constructing φ' can clearly be done in polynomial time (in the length of φ). By the accompanying arguments, we see that φ is satisfiable iff φ' is satisfiable. Thus we have CNF-SAT $\leq^p$ 3-SAT, and so, since 3-SAT is clearly in NP, 3-SAT is NP-complete.

3-SAT $\leq^p$ VERTEX COVER

The reduction from CNF-SAT to 3-SAT uses a technique called *local replacement*, where we take each piece of the input instance and replace it with some object constructed from the piece. In the reduction above, each piece was a clause, and we replaced it with one or more conjoined clauses.

To polynomially reduce 3-SAT to VERTEX COVER, we must show how, given an arbitrary instance φ of 3-SAT, to construct in polynomial time a graph G and integer K (depending on φ) such that φ is satisfiable if and only if G has a vertex cover of size at most K. We must construct G and K without knowing whether or not φ is satisfiable.

We use a different technique here, called *component design*. Given an instance φ of 3-SAT, we construct G out of two types of components: truth-setting components, which encode possible truth assignments of the variables of φ, and satisfaction-testing components—one for each clause—which check whether a clause is satisfied by a truth assignment.

Truth-Setting Components

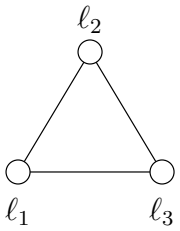
Let $u_1, \ldots, u_n$ be the variables of φ. G will have n truth-setting components—one for each variable. The component for u_i is a pair of vertices joined by an edge. We label one vertex u_i and the other $\overline{u_i}$. The n truth-setting components are pairwise disjoint, and there are no other edges between them.

Observe that any vertex cover of G must include at least one vertex in each component to cover its edge. If only n component vertices are in the cover, then each component has exactly one of

its two vertices in the cover. Such a minimum cover then corresponds to a truth assignment of the variables, namely, the one that makes each literal true iff it labels a vertex in the cover.

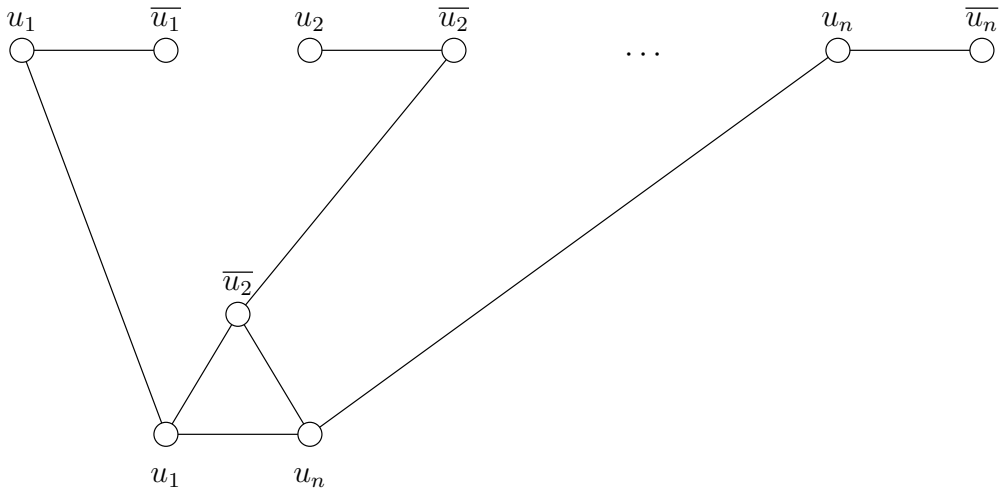
Clause-Satisfaction-Testing Components

Let $C_1, \ldots, C_m$ be the clauses of φ. For each clause $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$ of φ, the graph G will have a satisfaction-testing component consisting of three vertices joined by edges in a triangle. The vertices of the component are labeled by the literals ℓ_1, ℓ_2, and ℓ_3, respectively.



Observe that any vertex cover of G must include at least two vertices of each satisfaction-testing component. These components are pairwise disjoint, and there are no other edges between them.

Finally, we include an edge connecting each vertex in each satisfaction-testing component to the vertex in the truth-setting components with the same label. For example, for the claus $(u_1 \vee \overline{u_2} \vee u_n)$, we have



This concludes the construction of G. Setting $K = n + 2m$ completes the description of the output instance of VERTEX COVER constructed from φ. This construction is clearly doable in polynomial time.

Correctness

We need to show that φ is satisfiable if and only if the corresponding graph G has a vertex cover of size at most $n + 2m$. By the observations above, any vertex cover of G must have size at *least* $n + 2m$.

First, suppose φ is satisfiable. Let t be a truth assignment of the variables $u_1, \ldots, u_n$ that satisfies φ. Then there is a vertex cover A of G: A consists of exactly one vertex from each truth-setting component—the one labeled by the literal made true by t—together with exactly two vertices of each satisfaction-testing component, making $n + 2m$ vertices in all. The two vertices in

each satisfaction-testing component are chosen as follows: in the corresponding clause C_i, choose the leftmost literal made true by t (such a literal exists for each clause, because t satisfies φ), and include in A the two vertices labeled by the *other* two literals in C_i.

A has the right size. Is it a vertex cover for G? Each component's internal edges are covered by A, so we need only check that each edge connecting a truth-setting component to a satisfaction-testing component is covered. Each such edge e has endpoints labeled by the same literal ℓ. If ℓ is made true by t, then the endpoint of e in the truth-setting component is in A. If ℓ is made false by t, then the endpoint of e in the satisfaction-testing component must be in A, since the only literal of that component left out of A is made true by t. In either case, e is covered by A. Thus, A is a vertex cover.

Now we must argue the converse: if G has a vertex cover of size $n + 2m$, then φ is satisfiable. [Occasionally (not in this case) it is more convenient to argue the contrapositive, namely, if φ is *not* satisfiable, then *no* vertex cover of size $n+2m$ exists in G. A conditional statement, "if P then Q" is always logically equivalent to its contrapositive, "if not Q then not P."]

Suppose G has a vertex cover A of size $n+2m$. Then by the observations above, A must contain exactly one vertex of each truth-setting component and exactly two vertices of each satisfaction-testing component. Let t be the truth assignment of $u_1, \ldots, u_n$ that makes each literal true iff that literal labels a truth-setting vertex in A. The claim is that t satisfies φ. Let C_i be any clause of φ and let s_i be the satisfaction-testing component for C_i. One vertex u in s_i, labeled with some literal ℓ, is not in A. But the edge connecting u with the vertex v labeled ℓ in the truth-setting components must be covered by A, and so $v \in A$, and so by the definition of the truth assignment t, ℓ is made true by t, and thus C_i is satisfied. This is true for each clause, so t indeed satisfies φ.

This concludes the proof that 3-SAT $\leq^p$ VERTEX COVER, and hence VERTEX COVER is NP-complete.

INDEPENDENT SET and CLIQUE

Let $G = (V, E)$ be an undirected graph. A *clique* in G is a subset $C \subseteq V$ such that every two vertices in C are adjacent, i.e., C is a complete subgraph of G. An *independent set* in G is the opposite: a subset $I \subseteq V$ such that no two vertices in I are adjacent.

We can form decision problems based on the existence of cliques and independent sets.

CLIQUE
Instance: an undirected graph G and an integer $K > 0$.
Question: does G have a clique of size at least K?

INDEPENDENT SET
Instance: an undirected graph G and an integer $K > 0$.
Question: does G have an independent set of size at least K?

Both these problems are clearly in NP, and we'll show that both are NP-complete by polynomially reducing VERTEX COVER to INDEPENDENT SET and INDEPENDENT SET to CLIQUE. These problems are often useful for showing other problems to be NP-complete.

If $G = (V, E)$ is an undirected graph, we define G^c, the *complement* of G, to be (V, E'), where

$$E' = \{(u, v) \mid u, v \in V,\ u \neq v, \text{ and } (u, v) \notin E\}.$$

Note the following two easy facts:

1. A is a vertex cover in G iff $V - A$ is an independent set in G.

2. C is an independent set in G iff C is a clique in G^c.

Fact 1 implies that VERTEX COVER $\leq^p$ INDEPENDENT SET via the polynomial reduction that maps an instance G, K of VERTEX COVER to the instance G, K' of INDEPENDENT SET, where $K' = |V| - K$.

Fact 2 implies that INDEPENDENT SET $\leq^p$ CLIQUE via the polynomial reduction that maps an instance G, K of INDEPENDENT SET to the instance G^c, K of CLIQUE.

Restriction

The simplest technique for showing a decision problem Π NP-complete—one that works in many cases—is to show that there is a restriction of Π that is already known to be NP-complete. (One must of course also show that $\Pi \in \mathbf{NP}$.)

Definition 35 *A decision problem Π_1 is a* restriction *of a decision problem Π_2 if every yes-instance of Π_1 is a yes-instance of Π_2 and every no-instance of Π_1 is a no-instance of Π_2.*

For example, 3-SAT is a restriction of CNF-SAT, which is itself a restriction of SAT.

Fact 36 *If Π_1 is a restriction of Π_2, Π_1 is NP-complete, and $\Pi_2 \in \mathbf{NP}$, then Π_2 is NP-complete.*

Proof Π_1 polynomially reduces to Π_2 just by mapping each instance of Π_1 to itself. □

We can loosen the definition of restriction somewhat by allowing Π_1 to embed into Π_2 via some trivial transformation. Then Fact 36 still holds for this looser definition: the polynomial reduction performs this trivial transformation.

For example, the problem

> HAMILTONIAN PATH
> Instance: an undirected graph G.
> Question: is there a Hamiltonian path in G, i.e., a simple path that includes each vertex of G exactly once?

embeds trivially into the problem

> LONG PATH
> Instance: an undirected graph G and an integer $L > 0$.
> Question: does G have a simple path of length at least L?

by taking an instance $G = (V, E)$ of HAMILTONIAN PATH and mapping it to the instance $G, (|V| - 1)$ of LONG PATH. (Recall that we define the length of a path to be the number of edges in the path, which is one less than the number of vertices. A path is *simple* if no vertex appears more than once in the path.)

Supplemental Lecture
Faster Scalar and Matrix Multiplication

Integer Addition and Multiplication

We represent natural numbers in binary as usual. The size of a number n is the number of binary digits needed to represent n, namely, $\lceil \lg(n+1) \rceil = \lg n + O(1)$.

Adding two k-bit natural numbers takes time $\Theta(k)$, and the usual "add-and-carry" method that we all know from grade school (except that it is base 2) is aymptotically optimal.

```
Add(a, b, n)
// a[(n − 1)...0] and b[(n − 1)...0] are two bit arrays of length n.
// Returns a bit array s[n...0] of size n + 1.
    c←0
    for i←0 to n − 1 do
        s[i]←a[i] ⊕ b[i] ⊕ c // single-bit xor
        c← Majority(a[i], b[i], c)
    s[n]←c
    return s
```

Subtraction is also clearly linear time, using the "subtract-and-borrow" method from grade school.

Now consider multiplication. The grade school approach (adapted to base 2) is to repreatedly multiply the first number by each bit of the second, then add up all the results, suitably shifted.

```
Multiply(a, b, n)
// a[(n − 1)...0] and b[(n − 1)...0] are two bit arrays of length n
// s is a bit array of size 2n.
    pad a and b with leading 0s to length 2n − 1
    fill s with all 0s
    for i←0 to n − 1 do
        if b[i] = 1 then
            s← Add(s, A · 2^i, 2n − 1) // using arithmetic shift
    return S
```

Clearly, this algorithm takes time $\Theta(n^2)$. Can we do better?

A Divide-and-Conquer Approach to Multiplication

Assume that n is a power of 2. (Otherwise, pad input numbers with leading zeros out to the next higher power of 2.) We are given two n-bit natural numbers a and b to multiply.

Base case: $n = 1$. This is trivial.

Recursive case: $n = 2m$, where $m > 0$ is an integer (the next lower power of 2):

1. Write $a = a_h 2^m + a_\ell$ and $b = b_h 2^m + b_\ell$, where a_h and a_ℓ are unique integers in the range $0 \ldots 2^m - 1$. Similarly for b_h and b_ℓ. These are the high and low halves of a and b, respectively. Clearly,
$$ab = a_h b_h 2^n + (a_h b_\ell + a_\ell b_h) 2^m + a_\ell b_\ell.$$

2. Recursively compute the four subproducts $a_h b_h, a_h b_\ell, a_\ell b_h, a_\ell b_\ell$.

3. Add up the four subproducts, suitably shifted.

The time $T(n)$ of this algorithm satisfies $T(n) = 4T(n/2)+\Theta(n)$, and so by the Master Theorem, $T(n) = \Theta(n^2)$—no better than the grade school approach.

We can use a trick, however, to reduce the number of recursive multiplications from four to three. We recursively compute $a_h b_h$ and $a_\ell b_\ell$ as before, but then we recursively compute $p = (a_h + a_\ell)(b_h + b_\ell)$. This is enough, because we have

$$ab = a_h b_h 2^n + (p - a_h b_h - a_\ell b_\ell)2^m + a_\ell b_\ell.$$

Which requires a constant number of shifts, additions, and subtractions.

[There's a technical glitch here. The number of bits in $a_h + a_\ell$ and $b_h + b_\ell$ may be $m+1$ instead of m, so in multiplying these directly we lose our assumption that m is a power of two. There's no problem with adapting the algorithm for any n: just split each number into two numbers of $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ bits, respectively. This works best in practice, and it does not change the asympototic run time. Alternatively, we can keep the size of the recursive multiply to m-bit numbers as follows: to multiply two numbers x, y of $m + 1$ bits each, express

$$\begin{aligned} x &= b_x 2^m + c_x, \\ y &= b_y 2^m + c_y, \end{aligned}$$

where $b_x, b_y \in \{0, 1\}$ and c_x and c_y are expressible with m bits each (i.e., $0 \le c_x, c_y < 2^m$), then notice that

$$xy = b_x b_y 2^{2m} + (b_x c_y + b_y c_x)2^m + c_x c_y.$$

Only one product on the right-hand side is nontrivial ($c_x c_y$), which is a product of two m-bit integers.]

The running time $T(n)$ of the algorithm above now satisfies $T(n) = 3T(n/2) + \Theta(n)$, and so $T(n) = \Theta(n^{\lg 3}) = \Theta(n^{1.58\cdots})$, which is a significant speed-up over the naive quadratic time algorithm.

Matrix Multiplication

We can use a similar idea to multiply two $n \times n$ matrices faster than the obvious $\Theta(n^3)$ algorithm. This faster algorithm is due to Volker Strassen. Although the idea is similar, the actual algorithm is much more complicated in the case of matrices.

We use the divide-and-conquer approach again, assuming that n is a power of two. We count scalar multiplications, which we regard as primitive, and which dominate all the other operations (including addition). If $n = 1$, then there is only one scalar multiplication. Assume that $n > 1$ and let $m = n/2$. Suppose A and B are the two $n \times n$ matrices to be multiplied together to get an $n \times n$ matrix $C = AB$. We can chop each of A, B, and C into four $m \times m$ submatrices ("blocks") as follows:

$$\left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \cdot \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right] = \left[\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right].$$

The $m \times m$ submatrices are $a, b, c, d, e, f, g, h, r, s, t, u$. A nice property of matrix multiplication is that we can just multiply the submatrices as if they were scalars:

$$\begin{aligned} r &= ae + bg, \\ s &= af + bh, \\ t &= ce + dh, \\ u &= cf + dh. \end{aligned}$$

The only caveat is that matrix multiplication is not commutative, so the order of the factors matters.

We can naively just compute each of the eight products on the right-hand sides recursively then add up the results. This gives a running time $T(n)$ satisfying $T(n) = 8T(n/2) + \Theta(n^2)$, that is, $T(n) = \Theta(n^3)$ by the Master Theorem. This is asymptotically no better than our original approach.

By multiplying certain sums and differences of submatrices, Strassen found a way to reduce the number of recursive matrix multiplications from eight to seven. The run time of his method thus satisfies $T(n) = 7T(n/2) + \Theta(n^2)$, that is, $T(n) = \Theta(n^{\lg 7})$. I'll give his algorithm here so that you can check that it is correct and implement it if need be, but the presentation won't give much insight into how he came up with it.

Letting A, B, C and$n = 2m$ as above, we define $m \times m$ matrices A_i and B_i and $P_i = A_i B_i$, for $1 \leq i \leq 7$.

1. Let $A_1 = a$ and $B_1 = f - h$, and so $P_1 = af - ah$.
2. Let $A_2 = a + b$ and $B_2 = h$, and so $P_2 = ah + bh$ (whence $s = P_1 + P_2$).
3. Let $A_3 = c + d$ and $B_3 = e$, and so $P_3 = ce + de$.
4. Let $A_4 = d$ and $B_4 = g - e$, and so $P_4 = dg - de$ (whence $t = P_3 + P_4$).
5. Let $A_5 = a + d$ and $B_5 = e + h$, and so $P_5 = ae + ah + de + dh$.
6. Let $A_6 = b - d$ and $B_6 = g + h$, and so $P_6 = bg + bh - dg - dh$ (whence $r = P_5 + P_4 - P_2 + P_6$).
7. Let $A_7 = a - c$ and $B_7 = e + f$, and so $P_7 = ae + af - ce - cf$ (whence $u = P_5 + P_1 - P_3 - P_7$).

Exercise: verify that all the conclusions are correct.

Faster Algorithms

n-bit integer multiplication can actually be done in time $O(n \lg n)$, using a fast implementation of the Discrete Fourier Transform (DFT) known as the Fast Fourier Transform (FFT). The technique also works for multiplying two polynomials of degree n in time $O(n \lg n)$.

DFT over $\mathbb{Z}_n$

Fix an integer $n \geq 1$. Let $\omega_n = e^{2\pi i/n}$ (we call ω_n the *primitive nth root of unity*, because n is the least positive integer such that $\omega_n^n = 1$). The *Discrete Fourier Transform on* $\mathbb{Z}_n$, denoted DFT_n, is the linear map from $\mathbb{C}^n$ to $\mathbb{C}^n$ defined by the $n \times n$ matrix whose (i, j)th entry is $\omega_n^{ij}/\sqrt{n}$, for all $0 \leq i, j < n$. That is,

$$\mathrm{DFT}_n = \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2n-2} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3n-3} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2n-2} & \omega_n^{3n-3} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}.$$

It is easily checked that the inverse transformation, DFT_n^{-1}, is given by the matrix with (i, j)th entry equal to $\omega_n^{-ij}/\sqrt{n}$.

[The Fourier Transform is useful for a variety of purposes, including signal and image processing, because it decomposes a signal into its component pure frequencies. The digital signal read off of

an audio CD is in the frequency domain. A CD player applies an (inverse) Fourier Transform to produce an analog signal. Your ear is a natural Fourier transformer, allowing you to recognize the various pitches in music.]

Applying DFT_n to a vector of n complex numbers naively takes n^2 complex scalar multiplications. However, due to the extreme regularity of the DFT_n matrix, the time to apply both DFT_n and DFT_n^{-1} can be reduced to $O(n \lg n)$ in the case where n is a power of two, counting operations on complex scalars as primitive, which is reasonable for our purposes, since the we only need $O(\lg n)$ bits of precision. This is the Fast Fourier Transform (FFT), which we won't go into here.

To multiply two n-bit numbers a and b, where n is a power of 2, we first pad a and b with leading zeros out to $2n$ bits. Next, we apply DFT_{2n} separately to both a and b (as vectors of $2n$ values in $\{0, 1\}$), obtaining two vectors $\widehat{a}$ and $\widehat{b}$ of $2n$ complex numbers, respectively. Maintaining $\Theta(\lg n)$ bits of precision, compute a vector $\widehat{c}$ of size $2n$ whose ith entry is the product of the ith entries of $\widehat{a}$ and of $\widehat{b}$. Now apply DFT_{2n}^{-1} to $\widehat{c}$ to obtain a vector c' of $2n$ complex numbers. If we did this with infinite precision, the entries of c' would be all integers. With our level of precision, they are all close to integers. Let $\langle c_0, c_1, \ldots, c_{2n-1} \rangle$ be the vector such that c_i is the integer closest to the ith component of c'. Then

$$ab = \sum_{i=0}^{2n-1} c_i 2^i.$$

This last sum can be computed in time $O(n)$ using a method called Horner's Rule. The entire time taken to multiply a by b is thus $O(n \lg n)$ when we use the FFT to implement DFT.

FFT can be efficiently parallelized: it can be computed by a circuit (known as a "butterfly network") of depth $\lg n$ with a linear number of gates.

Fast Matrix Multiplication

The fastest known way to multiply two $n \times n$ matrices is due to Coppersmith and Winograd, and takes time $O(n^{2.376\ldots})$. Although this is the best known to date, the possibility of a faster algorithm has not been ruled out mathematically. For all we know, a linear time algorithm (i.e., one that runs in time $\Theta(n^2)$ because the inputs and outputs are both of size $\Theta(n^2)$) may exist.